NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCES
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

PROGRAM OF POSTGRADUATE STUDIES

PhD THESIS

A GPU performance estimation model based on micro-benchmarks and black-box kernel profiling

Elias N. Konstantinidis

ATHENS

JULY 2017

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΡΟΓΡΑΜΜΑ ΜΕΤΑΠΤΥΧΙΑΚΩΝ ΣΠΟΥΔΩΝ

ΔΙΔΑΚΤΟΡΙΚΗ ΔΙΑΤΡΙΒΗ

Ένα μοντέλο εκτίμησης απόδοσης επεξεργαστή γραφικών (GPU) βασισμένο σε μετροπρογράμματα και καταγραφή μετρικών με προσέγγιση «μαύρο-κουτί»

Ηλίας Ν. Κωνσταντινίδης

ΑΘΗΝΑ

ΙΟΥΛΙΟΣ 2017

PhD THESIS

A GPU performance estimation model based on micro-benchmarks and black-box kernel profiling

Elias N. Konstantinidis

SUPERVISOR: Yiannis Cotronis, Associate Professor NKUA

THREE-MEMBER ADVISORY COMMITTEE:

Yiannis Cotronis, Associate Professor NKUA

Elias Manolakos, Professor NKUA

Nectarios Koziris, Professor NTUA

SEVEN-MEMBER EXAMINATION COMMITTEE

Yiannis Cotronis, **Associate Professor NKUA**	**Elias Manolakos,** **Professor NKUA**
Nectarios Koziris, **Professor NTUA**	**Nikolaos Missirlis,** **Professor NKUA**
Dimitris Gizopoulos, **Professor NKUA**	**Dimitrios Soudris,** **Associate Professor NTUA**
Filippos Tzaferis, **Assistant Professor NKUA**	

Examination Date: July 3, 2017

ΔΙΔΑΚΤΟΡΙΚΗ ΔΙΑΤΡΙΒΗ

Ένα μοντέλο εκτίμησης απόδοσης επεξεργαστή γραφικών (GPU) βασισμένο σε μετροπρογράμματα και καταγραφή μετρικών με προσέγγιση «μαύρο-κουτί»

Ηλίας Ν. Κωνσταντινίδης

ΕΠΙΒΛΕΠΩΝ ΚΑΘΗΓΗΤΗΣ: Ιωάννης Κοτρώνης, Αναπληρωτής Καθηγητής ΕΚΠΑ

ΤΡΙΜΕΛΗΣ ΕΠΙΤΡΟΠΗ ΠΑΡΑΚΟΛΟΥΘΗΣΗΣ:

Ιωάννης Κοτρώνης, Αναπληρωτής Καθηγητής ΕΚΠΑ

Ηλίας Μανωλάκος, Καθηγητής ΕΚΠΑ

Νεκτάριος Κοζύρης, Καθηγητής ΕΜΠ

ΕΠΤΑΜΕΛΗΣ ΕΞΕΤΑΣΤΙΚΗ ΕΠΙΤΡΟΠΗ

Ιωάννης Κοτρώνης,
Αναπληρωτής Καθηγητής ΕΚΠΑ

Ηλίας Μανωλάκος,
Καθηγητής ΕΚΠΑ

Νεκτάριος Κοζύρης,
Καθηγητής ΕΜΠ

Νικόλαος Μισυρλής,
Καθηγητής ΕΚΠΑ

Δημήτρης Γκιζόπουλος,
Καθηγητής ΕΚΠΑ

Δημήτριος Σούντρης,
Αναπληρωτής Καθηγητής ΕΜΠ

Φίλιππος Τζαφέρης,
Επίκουρος Καθηγητής ΕΚΠΑ

Ημερομηνία Εξέτασης: 3 Ιουλίου 2017

ABSTRACT

Over the last decade GPUs have been established in the High Performance Computing sector as compute accelerators. The primary characteristics that justify this modern trend are the exceptionally high compute throughput and the remarkable power efficiency of GPUs. However, GPU performance is highly sensitive to many factors, e.g. the type of memory access patterns, branch divergence, the degree of parallelism and potential latencies. Consequently, the execution time of a kernel on a GPU is a difficult to predict measure. Unless the kernel is latency bound, a rough estimate of the execution time on a particular GPU could be provided by applying the roofline model, which is used to map the program's operation intensity to the peak expected performance on a particular processor. Though this approach is straightforward, it cannot not provide accurate prediction results.

In this thesis, after validating the roofline principle on GPUs by employing a micro-benchmark, an analytical throughput oriented performance model is proposed. In particular, this improves on the roofline model following a quantitative approach and a completely automated GPU performance prediction technique is presented. In this respect, the proposed model utilizes micro-benchmarking and profiling in a "*black-box*" fashion as no inspection of source/binary code is required. The proposed model combines GPU and kernel parameters in order to characterize the performance limiting factor and to predict the execution time on target hardware, by taking into account the efficiency of beneficial computational instructions. In addition, the "*quadrant-split*" visual representation is proposed, which captures the characteristics of multiple processors in relation to a particular kernel.

The experimental evaluation combines test executions on stencil computations (red/black SOR, LMSOR), matrix multiplication (SGEMM) and a total of 28 kernels of the Rodinia benchmark suite, all applied on six CUDA GPUs. The observed absolute error in predictions was 27.66% in the average case. Special cases of mispredicted results were investigated and justified. Moreover, the aforementioned micro-benchmark was used as a subject for performance prediction and the exhibited results were very accurate. Furthermore, the performance model was also examined in a cross vendor configuration by applying the prediction method on the same kernel source codes through the HIP programming environment supported on the AMD ROCm platform. Prediction errors were comparable to CUDA experiments despite the significant architectural differences evident between different vendor GPUs.

SUBJECT AREA: Parallel computing

KEYWORDS: performance model, GPU, roofline model

ΠΕΡΙΛΗΨΗ

Κατά την τελευταία δεκαετία, οι επεξεργαστές γραφικών (GPUs) έχουν εδραιωθεί στον τομέα των υπολογιστικών συστημάτων υψηλής απόδοσης ως επιταχυντές υπολογισμών. Τα βασικά χαρακτηριστικά που δικαιολογούν αυτή τη σύγχρονη τάση είναι η εξαιρετικά υψηλή υπολογιστική απόδοση τους και η αξιοσημείωτη ενεργειακή αποδοτικότητα τους. Ωστόσο, η απόδοση τους είναι πολύ ευαίσθητη σε πολλούς παράγοντες, όπως π.χ. τον τύπο των μοτίβων πρόσβασης στη μνήμη (memory access patterns), την απόκλιση διακλαδώσεων (branch divergence), τον βαθμό παραλληλισμού και τις δυνητικές καθυστερήσεις (latencies). Συνεπώς, ο χρόνος εκτέλεσης ενός πυρήνα (kernel) σε ένα επεξεργαστή γραφικών είναι ένα δύσκολα προβλέψιμο μέγεθος. Στην περίπτωση που η απόδοση του πυρήνα δεν περιορίζεται από καθυστερήσεις, μπορεί να παρασχεθεί μια χονδρική εκτίμηση του χρόνου εκτέλεσης σε ένα συγκεκριμένο επεξεργαστή εφαρμόζοντας το μοντέλο γραμμής-οροφής (roofline), το οποίο χρησιμοποιείται για να αντιστοιχίσει την ένταση υπολογισμών του προγράμματος στην μέγιστη αναμενόμενη απόδοση για ένα συγκεκριμένο επεξεργαστή. Αν και αυτή η προσέγγιση είναι απλή, δεν μπορεί να παρέχει ακριβή αποτελέσματα πρόβλεψης.

Σε αυτή τη διατριβή, μετά την επαλήθευση της αρχής του μοντέλου γραμμής-οροφής σε επεξεργαστές γραφικών με τη χρήση ενός μικρο-μετροπρογράμματος, προτείνεται ένα αναλυτικό μοντέλο απόδοσης. Συγκεκριμένα, βελτιώνεται το μοντέλο γραμμής-οροφής ακολουθώντας μια ποσοτική προσέγγιση και παρουσιάζεται μία πλήρως αυτοματοποιημένη μέθοδος πρόβλεψης απόδοσης σε επεξεργαστή γραφικών. Από αυτή την άποψη, το προτεινόμενο μοντέλο χρησιμοποιεί την αξιολόγηση μέσω μικρο-μετροπρογραμμάτων και την καταγραφή μετρικών με μέθοδο «*μαύρου κουτιού*», καθώς δεν απαιτείται διερεύνηση του πηγαίου/δυαδικού κώδικα. Το προτεινόμενο μοντέλο συνδυάζει τις παραμέτρους του επεξεργαστή γραφικών και του πυρήνα για να χαρακτηρίσει τον παράγοντα περιορισμού της απόδοσης και να προβλέψει το χρόνο εκτέλεσης στο στοχευόμενο υλικό, λαμβάνοντας υπόψη την αποδοτικότητα των ωφελίμων υπολογιστικών εντολών. Επιπλέον, προτείνεται η οπτική αναπαράσταση «*διαμοιρασμού-τεταρτημορίου*» (“*quadrant-split*”), η οποία αποδίδει τα χαρακτηριστικά πολλών επεξεργαστών σε σχέση με έναν συγκεκριμένο πυρήνα.

Η πειραματική αξιολόγηση συνδυάζει δοκιμαστικές εκτελέσεις σε υπολογισμούς μορίων (κόκκινο/μαύρο SOR, LMSOR), πολλαπλασιασμό πινάκων (SGEMM) και ένα σύνολο 28 πυρήνων της σουίτας μετροπρογραμμάτων Rodinia, όλα εφαρμοσμένα σε έξι επεξεργαστές γραφικών CUDA. Το παρατηρηθέν απόλυτο σφάλμα στις προβλέψεις ήταν 27,66% στη μέση περίπτωση. Διερευνήθηκαν και αιτιολογήθηκαν ιδιαίτερες περιπτώσεις εσφαλμένων προβλέψεων. Επιπλέον, το προαναφερθέν μικρο-μετροπρόγραμμα χρησιμοποιήθηκε ως αντικείμενο για την πρόβλεψη απόδοσης και τα αποτελέσματα ήταν πολύ ακριβή. Προσθέτως, το μοντέλο απόδοσης εξετάστηκε σε σύνθετο περιβάλλον μεταξύ διαφορετικών κατασκευαστών, εφαρμόζοντας τη μέθοδο πρόβλεψης στους ίδιους πηγαίους κώδικες πυρήνων μέσω του περιβάλλοντος προγραμματισμού HIP που υποστηρίζεται από την πλατφόρμα AMD ROCm. Τα σφάλματα πρόβλεψης ήταν συγκρίσιμα αυτών των πειραμάτων του περιβάλλοντος CUDA, παρά τις σημαντικές διαφορές αρχιτεκτονικής που παρατηρούνται μεταξύ των διαφορετικών κατασκευαστών επεξεργαστών γραφικών.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Παράλληλος υπολογισμός

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: μοντέλο απόδοσης, Μονάδα Επεξεργασίας Γραφικών, μοντέλο γραμμή-οροφής

ACKNOWLEDGEMENTS

The accomplishment of this thesis would not be possible without the great assistance and guidance of my advisor, Dr. Yiannis Cotronis. He has always been encouraging and supportive on my every single step during all these years of my PhD. His support has been valuable even in cases when I needed psychological encouragement.

However, this thesis required significant support by more than one persons. I could never reach to success without the constant support and enduring patience of my loving wife, Gina Gelasi. In addition, Nikos, my two and a half years old son, is another important person in my life who inadvertently helped me reach my goal. I also want to thank my parents for their support and for shaping me up to the person I've become.

I also want to thank the rest of my advisory committee, Dr. Nektarios Kozyris and Dr. Elias Manolakos, who have been helpful with their advice and commentary. Their suggestions and guidance helped me improve the orientation of this thesis and correct my defense presentation in various aspects.

I also want to personally thank the generous support of the manager of the Radeon Open Compute department of AMD, Greg Stoner, for his support through the donation of the Radeon graphic cards used in the experiments of this thesis.

Finally, I would like to thank Dr. Nectarios Koziris and Dr. Dimitris Gizopoulos to use their NVidia Tesla M2050 and NVidia Tesla K20c installations, respectively, for the purpose of the experiments of this thesis.

ΣΥΝΟΠΤΙΚΗ ΠΑΡΟΥΣΙΑΣΗ ΤΗΣ ΔΙΔΑΚΤΟΡΙΚΗΣ ΔΙΑΤΡΙΒΗΣ

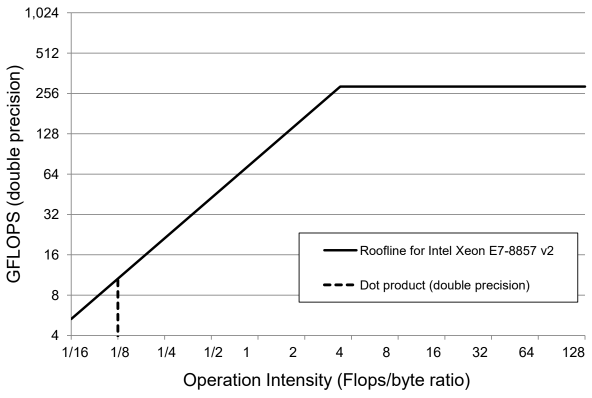
1. Εισαγωγή

Οι επεξεργαστές γραφικών (GPUs) στην σημερινή τους μορφή, εκτός από την κλασσική τους χρήση στην ρεαλιστική απεικόνιση τρισδιάστατων μοντέλων, έχουν υιοθετηθεί και για την επίλυση δύσκολων υπολογιστικών προβλημάτων γενικού σκοπού. Τα χαρακτηριστικά υψηλής απόδοσης για υπολογιστικό φόρτο μεγάλης παραλληλίας από τους επεξεργαστές γραφικών τους έχει καταστήσει ιδιαίτερα ελκυστικούς στην κοινότητα υπολογισμών υψηλής απόδοσης (High Performance Computing - HPC). Ταυτόχρονα, επιδεικνύουν υψηλούς δείκτες αποδοτικότητας σε σχέση με την ενεργειακή τους κατανάλωση. Για την εκμετάλλευσή τους αναπτύχθηκαν προγραμματιστικά περιβάλλοντα αμιγώς γενικού σκοπού. Οι παραπάνω λόγοι τους έχουν καταστήσει ως ιδανικούς επιταχυντές για την επίλυση μεγάλων υπολογιστικών προβλημάτων.

Η δημιουργία αυτού του νέου πεδίου εκμετάλλευσης των επεξεργαστών γραφικών δημιούργησε νέες ανάγκες στην μελέτη της απόδοσης τους, σε σχέση με την επίλυση υπολογιστικών προβλημάτων. Ωστόσο, ο προγραμματισμός τους τείνει να είναι επιρρεπής στην συχνή εμφάνιση προβλημάτων απόδοσης, συγκριτικά με τον προγραμματισμό των επεξεργαστών γενικού σκοπού. Η εμπειρία έχει δείξει ότι η απόδοση τους είναι πολύ πιο ευαίσθητη στην ορθή χρήση των πόρων και επομένως είναι πιο δύσκολο να εκτιμηθεί η τελική τους απόδοση. Για το σκοπό αυτό επινοήθηκαν διάφορα μοντέλα απόδοσης από την επιστημονική κοινότητα, προσανατολισμένα στις ιδιαιτερότητες των επεξεργαστών γραφικών. Πολλά από τα μοντέλα αυτά είναι αναλυτικής φύσεως, ενώ άλλα ακολουθούν την προσέγγιση της προσομοίωσης του επεξεργαστή γραφικών. Κάθε προσέγγιση έχει διαφορετικά πλεονεκτήματα και μειονεκτήματα. Η προσέγγιση της προσομοίωσης μπορεί να είναι εξαιρετικά ακριβής αλλά ταυτόχρονα και μια χρονοβόρος διαδικασία. Αντιθέτως, τα αναλυτικά μοντέλα τείνουν να είναι αρκετά γρηγορότερα στην εφαρμογή τους, ενώ παράλληλα τείνουν να παρέχουν καλύτερη αίσθηση πάνω στα χαρακτηριστικά υψηλού επιπέδου της εφαρμογής που μπορούν να επηρεάσουν τις επιδόσεις

Ένα από τα κλασσικά αναλυτικά μοντέλα απόδοσης είναι το μοντέλο γραμμής-οροφής (roofline). Το μοντέλο αυτό είναι ένα οπτικό μοντέλο που παρέχει ενόραση στην μέγιστη αναμενόμενη απόδοση ενός πυρήνα, λαμβάνοντας υπόψη τις ανάγκες τόσο σε καθαρούς υπολογισμούς, όσο και στην κίνηση από/προς τη μνήμη. Βασίζεται στην υπόθεση ότι περιοριστικό παράγοντα της απόδοσης αποτελεί είτε η ρυθμαπόδοση υπολογισμών, είτε το εύρος ζώνης της μνήμης του επεξεργαστή. Ο προσδιορισμός της εκάστοτε περίπτωσης προσδιορίζεται με βάση τις σχετικές απαιτήσεις πράξεων της εφαρμογής. Ο λόγος υπολογισμών προς μετακινήσεων μνήμης εκφράζεται με το μέτρο της *έντασης πράξεων* (operation intensity), το οποίο εκτιμάται σε μονάδες *flop/byte* και χρησιμοποιείται για τον προσδιορισμό του παράγοντα περιορισμού της απόδοσης σε ένα συγκεκριμένο επεξεργαστή. Η εκτίμηση της εντάσεως πράξεων ενός προγράμματος προσδιορίζεται μέσω της εξίσωσης (1). Για παράδειγμα, στο σχήμα 1 απεικονίζεται η γραμμή που εκφράζει τις προδιαγραφές ενός επεξεργαστή. Το διακεκομμένο κάθετο ευθύγραμμο τμήμα εκφράζει την ένταση πράξεων μιας εφαρμογής και το σημείο στο οποίο συναντά η τελευταία το γράφημα του επεξεργαστή ορίζει την μέγιστη αναμενόμενη απόδοση.

$$O_{kernel} = \frac{Operations_{(compute)}}{Traffic_{(memory)}} \tag{1}$$

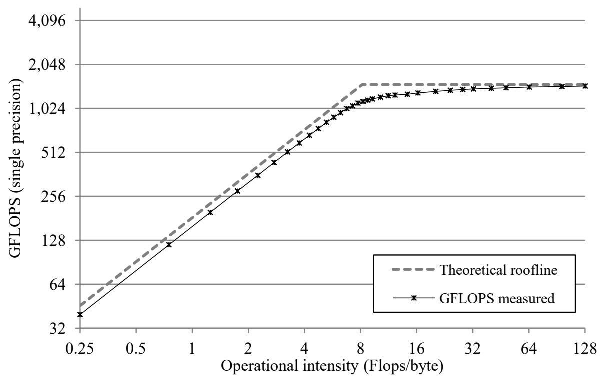


Σχήμα 1: Το μοντέλο γραμμής-οροφής για τον Intel Xeon E7-8857 v2.

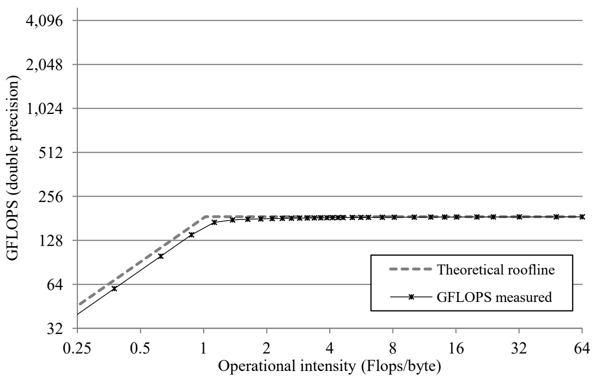
Το μοντέλο γραμμής-οροφής έχει εφαρμοστεί σε πληθώρα μελετών και εφαρμογών λόγω της απλότητάς του και της γενικότητάς του. Η ικανότητά του να αποδίδει με οπτικό τρόπο την μέγιστη δυνατή απόδοση ενός πυρήνα (kernel) σε ένα επεξεργαστή αποτελεί ένα από τα σημαντικότερα προτερήματα του. Ωστόσο, το μοντέλο είναι σε μεγάλο βαθμό αφαιρετικό και στις περισσότερες περιπτώσεις δεν επιτρέπει την διενέργεια ρεαλιστικών προσεγγίσεων παρά μόνο την εκτίμηση ενός άνω φράγματος απόδοσης.

2. Το μοντέλο γραμμής-οροφής σε επεξεργαστή γραφικών και η αναπαράσταση *διαμοιρασμού-τεταρτημορίου*

Αρχικά, σε αυτή τη διατριβή μελετάται η εφαρμοσιμότητα του μοντέλου γραμμής-οροφής στους επεξεργαστές γραφικών. Η μελέτη πραγματοποιείται μέσω ειδικού μετροπρογράμματος που αναπτύχθηκε για αυτό το σκοπό, το οποίο εκτιμά την απόδοση των επεξεργαστών γραφικών πειραματικά σε μικτό φόρτο υπολογισμών και μεταφορών μνήμης, σε ένα εύρος τιμών έντασης πράξεων. Μέσω του μετροπρογράμματος πραγματοποιήθηκε πειραματική προσέγγιση της θεωρητικής γραμμής-οροφής για 3 επεξεργαστές γραφικών, σε πράξεις κινητής υποδιαστολής μονής και διπλής ακρίβειας. Ενδεικτικά, στο σχήμα 2 απεικονίζεται η απόδοση του επεξεργαστή γραφικών GTX-480. Η διακεκομμένη γραμμή αναπαριστά την θεωρητική μέγιστη απόδοση με βάση τις προδιαγραφές του επεξεργαστή γραφικών. Οι παρατηρούμενη απόδοση ακολουθεί ένα παραπλήσιο μοτίβο της θεωρητικής απεικόνισης όπως περιγράφεται από την γραμμή-οροφής. Συγκεκριμένα, η απόδοση



(α΄) SP floating point

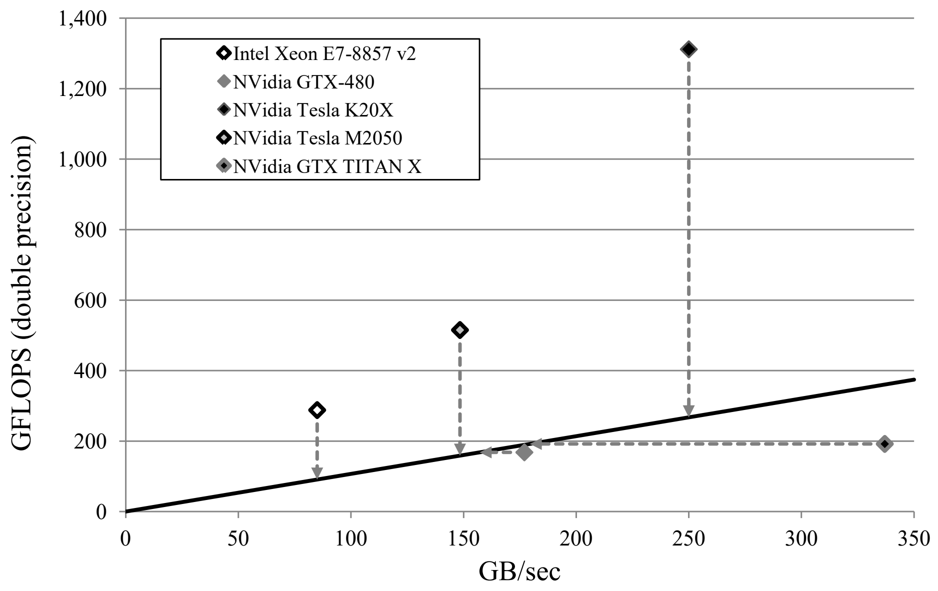


(β΄) DP floating point

Σχήμα 2: Πειραματική προσέγγιση της γραμμής-οροφής σε επ.γραφικών GTX-480

του επεξεργαστή γραφικών GTX-480 προσέγγισε σε μεγάλο βαθμό την θεωρητική μέγιστη υπολογιστική απόδοση (οριζόντιο τμήμα γραφήματος γραμμής-οροφής). Στο τμήμα που η απόδοση εξαρτάται από το εύρος ζώνης μνήμης (κεκλιμένο τμήμα γραφήματος γραμμής-οροφής), η απόδοση είναι ελαφρώς μειωμένη και το φαινόμενο αυτό εκφράζει την αδυναμία επίτευξης της μέγιστης θεωρητικής απόδοσης του εύρους ζώνης μνήμης σε πραγματικούς επεξεργαστές γραφικών. Παρόλο που η μέγιστη θεωρητική απόδοση του GTX-480 που χρησιμοποιήθηκε για τα πειράματα είναι 182GB/sec, το εύρος ζώνης μνήμης που μετρήθηκε δεν ξεπέρασε τα 163GB/sec. Γενικά, τα αποτελέσματα έδειξαν ικανοποιητική ταύτιση της θεωρητικής γραμμής-οροφής με την πειραματική. Παρατηρήθηκαν κάποιες αποκλίσεις, κυρίως στην επίτευξη του μεγίστου εύρους ζώνης μνήμης και προκύψαν παρατηρήσεις.

Μια εναλλακτική μέθοδος απεικόνισης που προτείνεται σε αυτή τη διατριβή είναι η απεικόνιση «*διαμοιρασμού-τεταρτημορίου*» (*quadrant-split*). Συγκριτικά με την απεικόνιση γραμμής-οροφής, ο οριζόντιος άξονας περιγράφει το εύρος ζώνης μνήμης αντί της εντάσεως πράξεων. Με αυτή τη διαφοροποίηση ο κάθε επεξεργαστής περιγράφεται σαν σημείο και η εφαρμογή ως μια ημι-ευθεία που διαπερνά το σημείο τομής των αξόνων και η κλίση της προσδιορίζεται από την τιμή της εντάσεως πράξεων της εφαρμογής. Για παράδειγμα, στο σχήμα 3 αναπαρίσταται το πρόβλημα LBMHD (Lattice-Boltzmann Magnetohydrodynamic) σε σχέση με 4 επεξεργαστές γραφικών και ένα επεξεργαστή γενικού σκοπού. Τα σημεία του επεξεργαστή Intel Xeon και των NVidia Tesla επεξεργαστών γραφικών εντοπίζονται πάνω από την ημι-ευθεία, το οποίο σημαίνει ότι στους συγκεκριμένους επεξεργαστές η απόδοση αναμένεται να καθορίζεται από το μέγιστο εύρος ζώνης μνήμης (memory bound). Σε αντίθεση, τα σημεία των επεξεργαστών γραφικών GTX-480 και GTX-Titan X εντοπίζονται κάτω από την ημι-ευθεία και συνεπώς η απόδοση τους αναμένεται να καθορίζεται από την μέγιστη ρυθμαπόδοση των επεξεργαστών (compute bound). Οι διακεκομμένες γραμμές υποδεικνύουν τα σημεία πάνω στην ημι-ευθεία του προβλήματος που αναπαριστούν την εκτιμώμενη απόδοση για κάθε επεξεργαστή στο συγκεκριμένο πρόβλημα. Για την εκτίμηση της απόδοσης των άνω σημείων πραγματοποιείται διάσχιση από το σημείο με κάθετη φορά προς τα κάτω, μέχρι την συνάντηση της ημι-ευθείας. Αντίστοιχα, για την εκτίμηση της απόδοσης των κάτω σημείων πραγματοποιείται διάσχιση από το σημείο με οριζόντια φορά προς τα αριστερά, μέχρι την συνάντηση της ημι-ευθείας. Η τομή των διακεκομμένων τμημάτων με την ημι-ευθεία της εφαρμογής υποδεικνύουν την αναμενόμενη απόδοση σε κάθε επεξεργαστή.



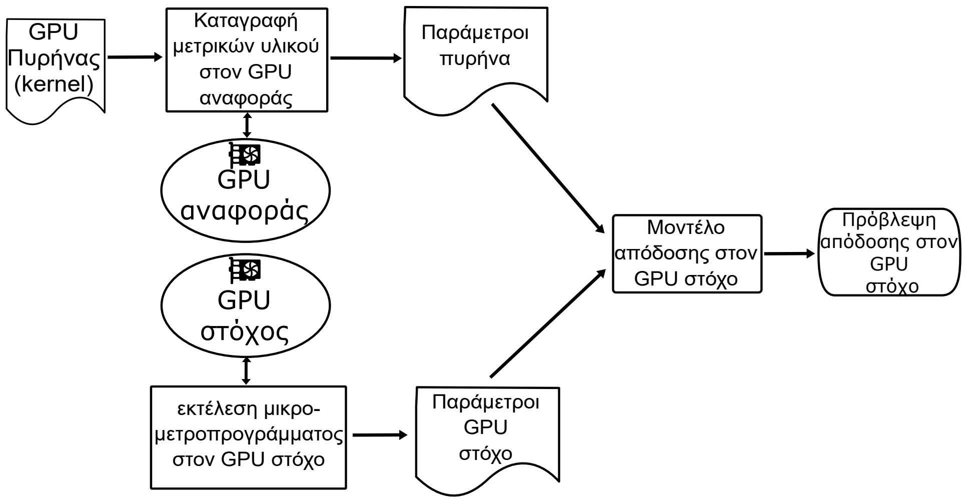
Σχήμα 3: Η αναπαράσταση *διαμοιρασμού-τεταρτημορίου* του προβλήματος LBMHD με εφαρμογή σε 5 επεξεργαστές (γραφικών και γενικού σκοπού).

Με την απεικόνιση *διαμοιρασμού-τεταρτημορίου* περισσότεροι του ενός επεξεργαστές μπορούν να αναπαρασταθούν με φυσικό τρόπο σε ένα μοναδικό γράφημα. Η γραφική εκτίμηση της θεωρητικής απόδοσης πραγματοποιείται με απλό και ευθύ τρόπο. Η αναπαράσταση γραμμής-οροφής είναι προσανατολισμένη στην απεικόνιση ενός επεξεργαστή σε σχέση με πολλά προβλήματα, ενώ η προτεινόμενη απεικόνιση προσφέρεται για την περίπτωση μίας εφαρμογής σε σχέση με πολλούς επεξεργαστές με διαφορετικά χαρακτηριστικά.

3. Ένα ποσοτικό μοντέλο απόδοσης για επεξεργαστές γραφικών

Βασική συνεισφορά της διατριβής αποτελεί ένα μοντέλο απόδοσης για επεξεργαστή γραφικών, που εστιάζει στην ποσοτική προσέγγιση. Αφού προηγήθηκε η παρατήρηση της ικανοποιητικής προσέγγισης του μοντέλου γραμμής-οροφής μέσω της εφαρμογής μικρομετροπρογράμματος, στη συνέχεια προέκυψε το ερώτημα σχετικά με το ποια χαρακτηριστικά πραγματικών εφαρμογών για επεξεργαστές γραφικών αποτρέπουν την επίτευξη μέγιστης απόδοσης, παραπλήσιας με αυτήν που αποδίδεται από το μοντέλο γραμμής-οροφής. Με αυτό το σκεπτικό, μελετήθηκαν τα χαρακτηριστικά βελτιστοποιημένων εφαρμογών τα οποία μπορούν να επηρεάσουν αρνητικά την απόδοση σε σημαντικό βαθμό.

Αυτή η μελέτη οδήγησε στον ορισμό ενός βελτιωμένου μοντέλου απόδοσης βασισμένο



GPU Πυρήνας (kernel)
Καταγραφή μετρικών υλικού στον GPU αναφοράς
Παράμετροι πυρήνα
GPU αναφοράς
GPU στόχος
Μοντέλο απόδοσης στον GPU στόχο
Πρόβλεψη απόδοσης στον GPU στόχο
εκτέλεση μικρο-μετροπρογράμματος στον GPU στόχο
Παράμετροι GPU στόχο

Σχήμα 4: Η ροή των βημάτων εκτέλεσης της μεθόδου πρόβλεψης απόδοσης.

στο μοντέλο γραμμής-οροφής, το οποίο παράλληλα εμβαθύνει ακολουθώντας ποσοτική προσέγγιση με ρεαλιστικές εκτιμήσεις πραγματικών εφαρμογών. Το διάγραμμα ροής των βημάτων όπως περιγράφονται από την προτεινόμενη μέθοδο απεικονίζεται στο σχήμα 4. Βασικά χαρακτηριστικά που ενσωματώθηκαν στο μοντέλο αποτελούν η αποδοτικότητα των ωφελίμων πράξεων, καθώς επίσης και η αποδοτικότητα του συνόλου εκτελούμενων εντολών ως προς το ποσοστό κατά το οποίο εκμεταλλεύονται τους εκτελεστικούς πόρους του επεξεργαστή γραφικών. Ωφέλιμες πράξεις θεωρούνται οι πράξεις που συνεισφέρουν άμεσα στην επίλυση του προβλήματος, δηλαδή τυπικά οι πράξεις κινητής υποδιαστολής. Ως αποδοτικότητα ωφελίμων πράξεων ορίζεται το στατιστικό μέγεθος που εκφράζει την αναμενόμενη τιμή του πλήθους πράξεων που πραγματοποιεί κάθε υπολογιστική εντολή του πυρήνα διαιρεμένο με το μέγιστο πλήθος πράξεων που μπορεί να πραγματοποιήσει μια υπολογιστική εντολή, τυπικά ο αριθμός 2, όπως προκύπτει από το πλήθος πράξεων που πραγματοποιεί μια εντολή τύπου πολλαπλασιασμού-πρόσθεσης (multiply-addition). Ως αποδοτικότητα του συνόλου εκτελούμενων εντολών ορίζεται το ποσοστό κατά το οποίο οι εκτελεστικοί πόροι του επεξεργαστή πραγματοποιούν εκτέλεση ωφελίμων εντολών αντί δευτερευόντων. Τέλος, η μέθοδος λαμβάνει υπόψη πραγματικές μετρήσεις των ρυθμαποδόσεων για ένα σύνολο προκαθορισμένων πράξεων και του εύρους ζώνης μνήμης με την χρήση μικρο-μετροπρογραμμάτων αντί των προδιαγραφών που δίνει ο κατασκευαστής.

Συγκεκριμένα, οι απαιτούμενες μετρικές υλικού που καταγράφονται κατά την εκτέλεση του πυρήνα παρέχονται στον πίνακα 1. Αυτές οι μετρικές απαιτούνται για την εκτίμηση των παραμέτρων του πυρήνα, οι οποίες παρουσιάζονται στον πίνακα 2. Ως κανόνας υπολογισμού για την εκτίμηση του K_{type} εφαρμόσθηκε η επιλογή του *fp64* εφόσον η μετρική M_{fp64} είναι μη μηδενική, του *fp32* αν η μετρική M_{fp32} είναι μη μηδενική ή διαφορετικά του *int*. Για

Πίνακας 1: Οι απαιτούμενες μετρικές υλικού του NVidia GPU για την εκτίμηση των παραμέτρων του πυρήνα.

Μετρική	Συμβολισμός
flop_count_sp_fma	M_{fma32}
flop_count_dp_fma	M_{fma64}
inst_compute_ld_st	M_{ldst}
inst_executed	M_{inst}
inst_fp_32	M_{fp32}
inst_fp_64	M_{fp64}
inst_integer	M_{int}
dram_read_transactions	$M_{tran\text{-}r}$
dram_write_transactions	$M_{tran\text{-}w}$

Πίνακας 2: Το σύνολο των απαιτούμενων παραμέτρων του πυρήνα.

Παράμετρος	Περιγραφή	Ανάκτηση
K_{type}	Τύπος κύριων πράξεων (fp64, fp32 ή int)	κανόνας υπολογισμού
W_{comp}	Πράξεις υπολογισμών	τύπος (2)
W_{traf}	Πλήθος προσπελαθέντων DRAM bytes	τύπος (3)
E_{mix}	Αποδοτικότητα μίξης πράξεων (%)	τύπος (4)
D_{ops}	Πυκνότητα εντολών πράξεων (%)	τύπος (7)
D_{ldst}	Πυκνότητα εντολών ανάκτησης/αποθήκευσης (%)	τύπος (8)
D_{other}	Πυκνότητα άλλων εντολών (%)	τύπος (9)

την εκτίμηση των παραμέτρων απαιτείται η εφαρμογή των παρακάτω τύπων:

$$W_{comp} = \begin{cases} M_{fp32} + M_{fma32}, & \text{αν } K_{type} = \text{fp32} \\ M_{fp64} + M_{fma64}, & \text{αν } K_{type} = \text{fp64} \\ M_{int}, & \text{αν } K_{type} = \text{int} \end{cases} \tag{2}$$

$$W_{traf} = 32 \times (M_{tran\text{-}r} + M_{tran\text{-}w}) \tag{3}$$

$$E_{mix} = \begin{cases} \frac{M_{fp32}+M_{fma32}}{2\times M_{fp32}} \times 100\%, & \text{αν } K_{type} = \text{fp32} \\ \frac{M_{fp64}+M_{fma64}}{2\times M_{fp64}} \times 100\%, & \text{αν } K_{type} = \text{fp64} \\ 50\%, & \text{αν } K_{type} = \text{int} \end{cases} \tag{4}$$

$$I_{ops} = \begin{cases} M_{fp32}, & \text{αν } K_{type} = \text{fp32} \\ M_{fp64}, & \text{αν } K_{type} = \text{fp64} \\ M_{int}, & \text{αν } K_{type} = \text{int} \end{cases} \tag{5}$$

$$I_{total} = 32 \times M_{inst} \tag{6}$$

$$D_{ops} = \frac{I_{ops}}{I_{total}} \times 100\% \tag{7}$$

$$D_{ldst} = \frac{M_{ldst}}{I_{total}} \times 100\% \tag{8}$$

$$D_{other} = 100\% - D_{ops} - D_{ldst} \tag{9}$$

Οι απαιτούμενοι παράμετροι του επεξεργαστή γραφικών με τις οποίες θα πραγματοποιηθεί η πρόβλεψη δίνονται στον πίνακα 3. Όλες οι παράμετροι εκτιμώνται με την εκτέλεση ειδικών μικρο-μετροπρογραμμάτων.

Πίνακας 3: Το σύνολο των παραμέτρων του επεξεργαστή γραφικών.

Παράμετρος	Περιγραφή	Μονάδα
T_{SP}	Ρυθμαπόδοση πρ.κινητής υποδιαστολής (μονής ακρίβειας)	GFLOPS
T_{DP}	Ρυθμαπόδοση πρ.κινητής υποδιαστολής (διπλής ακρίβειας)	GFLOPS
T_{int}	Ρυθμαπόδοση πρ.ακεραίων (πολλαπλασ.-προσθέσεων)	GIOPS
T_{add}	Ρυθμαπόδοση πράξεων ακεραίων (προσθέσεων)	GIOPS
T_{ldst}	Ρυθμαπόδοση εντ.ανάκτησης/αποθήκευσης διαμοιραζ.μν.	GOPS
B_{mem}	Εύρος ζώνης μνήμης	GB/sec

Η πρόβλεψη απόδοσης προϋποθέτει την εκτίμηση της αποδοτικότητας εντολών (E_{instr}) και στην συνέχεια την εκτιμώμενη μέγιστη ρυθμαπόδοση σε ωφέλιμες πράξεις (T'_{op}). Ο υπολογισμός πραγματοποιείται ως ακολούθως:

$$T_{op} = \begin{cases} T_{SP}, & \text{αν } K_{type} = \text{fp32} \\ T_{DP}, & \text{αν } K_{type} = \text{fp64} \\ T_{int}, & \text{αν } K_{type} = \text{int} \end{cases} \tag{10}$$

$$W_{op} = \frac{T_{SP}}{T_{op}} \tag{11}$$

$$W_{ldst} = \frac{{}^{1}\!/_{2} T_{SP}}{T_{ldst}} \tag{12}$$

$$W_{other} = \frac{{}^{1}\!/_{2} T_{SP}}{T_{add}} \tag{13}$$

$$C_{op} = D_{ops} \times W_{op} \tag{14}$$

$$C_{ldst} = D_{ldst} \times W_{ldst} \tag{15}$$

$$C_{other} = D_{other} \times W_{other} \tag{16}$$

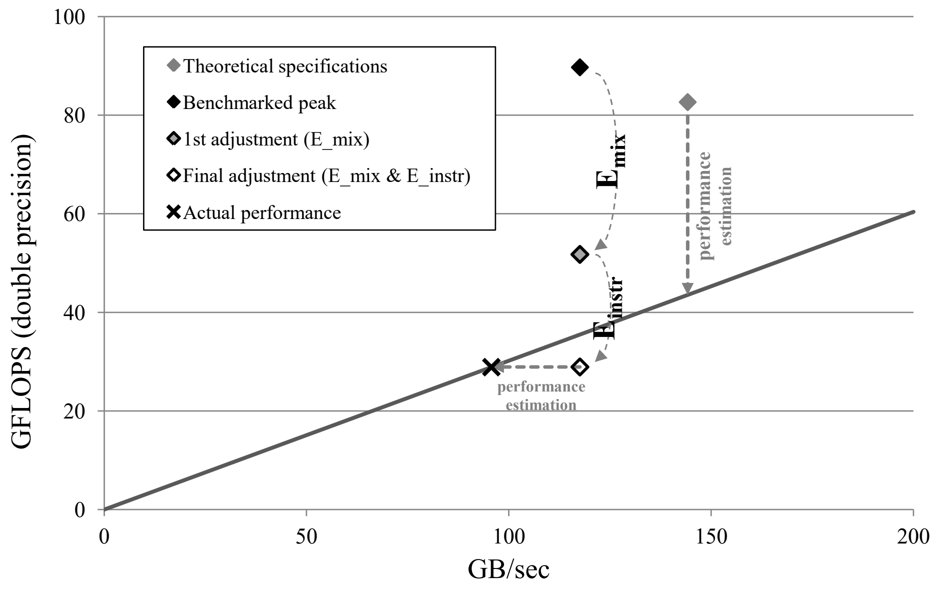
$$E_{instr} = \frac{C_{op}}{C_{op} + C_{ldst} + C_{other}} \times 100\% \tag{17}$$

$$T'_{op} = E_{mix} \times E_{instr} \times T_{op} \tag{18}$$

Στην τελική εκτιμώμενη απόδοση λαμβάνεται υπόψη και η χρήση του εύρους ζώνης μνήμης όπως φαίνεται στον παρακάτω τύπο:

$$T_{predicted} = \begin{cases} T'_{op}, & \text{αν } O_{krn} > O_{dev} \\ O_{krn} \times B_{mem}, & \text{αν } O_{krn} \leq O_{dev} \end{cases} \tag{19}$$

Η μέθοδος αναλύθηκε με την επίδειξη μελέτης περιπτώσεων στα προβλήματα υπολογισμού μορίων (κόκκινο/μαύρο SOR) και πολλαπλασιασμού πινάκων, με επεξεργαστή γραφικών αναφοράς τον GTX-480 και πρόβλεψη στον πάνω στον επεξεργαστή γραφικών GTX-660. Η πρόβλεψη με οπτική αναπαράσταση απεικονίζεται στο σχήμα 5. Τα αποτελέσματα επαληθεύτηκαν τόσο με την αντιπαραβολή τους με πραγματικές μετρήσεις όσο και με την ανάλυση τιμών μετρικών χρησιμοποίησης (utilization metrics) για τους διάφορους πόρους του επεξεργαστή γραφικών. Τέλος, μελετήθηκαν οι προϋποθέσεις κάτω από τις οποίες αναμένεται η μέθοδος να αποφέρει αξιόπιστα αποτελέσματα.



Σχήμα 5: Οπτικοποίηση της πρόβλεψης απόδοσης της μεθόδου κόκκινο/μαύρο SOR στον επεξεργαστή γραφικών GTX-660 με την εφαρμογή των προσαρμογών αποδοτικότητας.

4. Πειραματική αξιολόγηση

Στη συνέχεια ακολουθεί η πειραματική αξιολόγηση της μεθόδου. Επιλέχθηκε ένα ευρύ σύνολο εφαρμογών, αποτελούμενο από υπολογισμούς μορίων (stencils), πολλαπλασιασμό πινάκων (SGEMM) και 28 πυρήνες από 16 μετροπρογράμματα της σουίτας Rodinia. Οι εφαρμογές υπολογισμών μορίων περιελάμβαναν τις μεθόδους κόκκινο/μαύρο (red/black) SOR και LMSOR, οι υλοποιήσεις των οποίων αναπτύχθηκαν και μελετήθηκαν εκτενώς προγενέστερα κατά τη διάρκεια της εκπόνησης της διατριβής και όπως αποδείχθηκε πειραματικά, η αναδιάταξη των στοιχείων με βάση το χρώμα καθώς επίσης και η τακτική του επανυπολογισμού μπορούν να οδηγήσουν σε σημαντική επιτάχυνση της εκτέλεσης προβλημάτων με έντονη χρήση της μνήμης. Τα αποτελέσματα των προβλέψεων αναλύθηκαν και σε περιπτώσεις σημαντικής απόκλισης με τις πραγματικές μετρήσεις πραγματοποιήθηκε πρόσθετη ανάλυση για την αιτιολόγηση των αποτελεσμάτων. Στην πλειονότητα των περιπτώσεων (>50%) το απόλυτο σφάλμα πρόβλεψης εκτιμήθηκε κάτω του 25%, το οποίο και θεωρήθηκε αρκετά ικανοποιητικό. Επιπλέον, μελετήθηκε η ακρίβεια των αποτελεσμάτων του μοντέλου καθώς εφαρμόστηκε στο μικρο-μετροπρόγραμμα που αναφέρθηκε στο πρώτο μέρος. Τα αποτελέσματα ήταν εξαιρετικά ακριβή. Στην αρχική μορφή το μοντέλο περιορίστηκε στο περιβάλλον λογισμικού και υλικού της NVidia το οποίο αποτελεί προϋπόθεση για το περιβάλλον CUDA.

5. Αξιολόγηση μεταξύ διαφορετικών κατασκευαστών και περιορισμοί του μοντέλου

Σε μια δεύτερη μελέτη διερευνήθηκε η εφαρμοσιμότητα της μεθόδου σε επεξεργαστή γραφικών διαφορετικού κατασκευαστή. Το προγραμματιστικό περιβάλλον που επιλέχθηκε είναι το HIP, κάτω από την πλατφόρμα ROCm της AMD, το οποίο προσφέρει ένα γρήγορο και σχεδόν αυτοματοποιημένο τρόπο μετάβασης από το CUDA, διατηρώντας κατ ουσίαν τον κώδικα των πυρήνων αναλλοίωτο. Η ανάκτηση των τιμών των παραμέτρων των πυρήνων πραγματοποιήθηκε σε επεξεργαστή γραφικών της NVidia λόγω έλλειψης των αντιστοίχων μετρικών στους επεξεργαστές γραφικών της AMD. Η πρόβλεψη απόδοσης πραγματοποιήθηκε για τον επεξεργαστή γραφικών της AMD. Παρ όλες τις διαφορές των μεταξύ αρχιτεκτονικών τα αποτελέσματα πρόβλεψης στα 3 προγράμματα στα οποία εφαρμόστηκε η ανάλυση (μέθοδος κόκκινο/μαύρο SOR, πολλαπλασιασμός πινάκων και το lavaMD από την σουίτα Rodinia) έδειξαν ιδιαίτερα θετικά αποτελέσματα καθώς το απόλυτο σφάλμα εκτιμήθηκε στα ίδια πλαίσια στα οποία εκτιμήθηκε και στο περιβάλλον CUDA. Τέλος, πραγματοποιήθηκε εκτενής ανάλυση των εγγενών περιορισμών του μοντέλου και των ορίων του. Ακόμα περισσότερο, μελετήθηκε η εφαρμογή του μοντέλου σε δύο ακόμα μικρο-μετροπρογράμματα, τα οποία επίσης αναπτύχθηκαν στα πλαίσια της διατριβής. Το πρώτο προκαλεί έντονη χρήση των βοηθητικών μνημών (cache) του 1ου και 2ου επιπέδου και το δεύτερο προκαλεί έντονη χρήση της διαμοιραζόμενης μνήμης (shared memory) του επεξεργαστή γραφικών. Τα αποτελέσματα αναδεικνύουν δύο από τους περιορισμούς του μοντέλου που αναφέρθηκαν προηγουμένως με πειραματικό τρόπο. Σε γενικές γραμμές, το προτεινόμενο μοντέλο ενσωματώνει τα χαρακτηριστικά της αφαιρετικότητας και της ικανότητας ρεαλιστικής πρόβλεψης σε μια ιδανική ισορροπία.

6. Συμπεράσματα και μελλοντικές κατευθύνσεις

Σε αυτή τη διατριβή παρουσιάζεται ένα αναλυτικό μοντέλο απόδοσης, το οποίο βασίζεται στο μοντέλο γραμμής-οροφής. Η βασική αρχή του τελευταίου επαληθεύτηκε πειραματικά στους επεξεργαστές γραφικών με την ανάπτυξη εξειδικευμένου μικρο-μετροπρογράμματος, μέσω του οποίου η απόδοση διερευνήθηκε σε ένα ευρύ φάσμα τιμών εντάσεως πράξεων. Μέσω ποσοτικής προσέγγισης, το προτεινόμενο μοντέλο απόδοσης είναι ικανό να παρέχει προβλέψεις που προσεγγίζουν πραγματικούς χρόνους εκτέλεσης στο υλικό. Πρόσθετα, παρουσιάστηκε μια εναλλακτική οπτική αναπαράσταση, ονομαζόμενη «*διαμοιρασμού-τεταρτημορίου*», η οποίο παρέχει βελτιωμένη ενόραση σε περιπτώσεις όπου πολλοί επεξεργαστές αναπαριστώνται σε σχέση με μία εφαρμογή. Η αξία της απλότητας του μοντέλου και το υψηλό επίπεδο αφαιρετικότητας επιτρέπουν την παροχή αποτελεσμάτων, τα οποία μπορούν εύκολα να ερμηνευθούν και να αξιοποιηθούν από τον προγραμματιστή για την εξαγωγή πολύτιμων συμπερασμάτων.

Ένα από τα σημαντικά σημεία της προτεινόμενης μεθόδου είναι η ικανότητά της να εξαγάγει τις τιμές των παραμέτρων με την εκμετάλλευση ενός απλού συνόλου τιμών μετρικών υλικού. Η σύλληψη των τιμών του πραγματοποιείται με προσέγγιση «μαύρο-κουτί» καθώς δεν απαιτεί κάποια εσωτερική γνώση των εσωτερικών δομών του πυρήνα. Επιπλέον, η προτεινόμενη μέθοδος μπορεί να αναπτυχθεί ως ένα πλήρως αυτοματοποιημένο εργαλείο, το οποίο να λειτουργεί χωρίς την παρέμβαση του προγραμματιστή ή προηγούμενη διερεύνηση του κώδικα του πυρήνα.

Η μέθοδος επιτρέπει καλύτερη κατανόηση του φόρτου υπολογισμού και μνήμης σε αντίθεση με μια αμιγώς θεωρητική προσέγγιση κυρίως για δύο λόγους. Πρώτον, τόσο η εκτέλεση των μη ουσιαστικών εντολών όσο και των εντολών ανάκτησης/αποθήκευσης λαμβάνονται υπόψη μέσω της μοντελοποίησης των επιπτώσεων τους στον κορεσμό της διοχέτευσης εντολών. Πρόσθετα, ο τύπος της μίξης υπολογιστικών πράξεων λαμβάνεται επίσης υπόψη, δηλαδή η αποδοτικότητά της σε σχέση με το πλήθος πράξεων ανά εντολή. Δεύτερον, οι απαιτήσεις σε φόρτο προσβάσεων μνήμης εκτιμώνται μέσω της μέτρησης της πραγματικής κίνησης στη μνήμη, το οποίο σημαίνει ότι οι τοπικότητες καθώς επίσης και οι συγχωνεύσεις των προσβάσεων λαμβάνονται υπόψη εμμέσως στο μοντέλο.

Σαν μελλοντική κατεύθυνση προτείνεται η βελτίωση του μοντέλου με στόχο την καλύτερη ακρίβεια πρόβλεψης. Κάποιες βελτιώσεις θα μπορούσαν να περιλαμβάνουν την καλύτερη εκτίμηση του κόστους εκτέλεσης των εντολών, την ποσοτικοποίηση της απόκλισης εκτέλεσης των νημάτων, του μειωμένου παραλληλισμού λόγω άλλων φαινομένων όπως συγκρούσεων πρόσβασης σε διαμοιραζόμενη ή σταθερή μνήμη. Η μεταβλητότητα των φαινομένων λόγω βοηθητικής μνήμης είναι επίσης ένα ακόμα ζήτημα που θα ήταν σκόπιμο να ενσωματωθεί στο μοντέλο καθώς επηρεάζει το πλήθος των πραγματικών προσβάσεων που παρατηρούνται στην κύρια μνήμη. Ωστόσο, σε κάθε βελτίωση θα πρέπει να λαμβάνεται υπόψη η πιθανή διατάραξη της αφαιρετικότητας και του χαρακτήριστικού «μαύρο-κουτί» της μεθόδου.

To my loving family

CONTENTS

LIST OF FIGURES

LIST OF TABLES

PREFACE

I'm in the delightful position to author the finishing lines in this manuscript known as *PhD thesis*. The feelings are mixed. It's a great relief that after all the enormous and long effort spent, this work is finally reaching to its completion. On the other hand, there is a sense of emotion as this journey is approaching to the end. I would think of finishing of this thesis to be as one of the most important milestones in my whole life. I thank everyone cordially who helped me reaching this goal.

1. INTRODUCTION

Traditionally, the GPU (Graphics Processor Unit) has been used to accelerate the generation of stunning 3D image representations for 3D games and 3D professional graphics applications. Complex 3D image rendering requires a huge number of compute operations to be performed in order to create realistic images. In addition, the 3D rendering process is an embarrassingly parallel problem and this fact led to the adoption of parallel computation paradigm on GPUs in a much quicker pace and to a vastly larger extent than the traditional CPUs did. As GPUs steadily evolved to be more and more powerful rendering engines in recent years they surpassed the compute capabilities of CPUs. Thus, researchers began experimenting on general purpose computation on GPUs. Nowadays, they are considered to be alternative compute accelerators and have found their place in the HPC (High Performance Computing) sector which provides the necessary parallel compute workloads for GPU consumption. However, GPU programming tends to be more cumbersome and tedious compared to the CPU programming posing performance bottlenecks. Moreover, experience has shown that performance is much more sensitive to the proper use of GPU resources and thus it is more difficult to predict.

1.1 General purpose computation on GPUs

Historically, a graphics accelerator which afterwards evolved to a GPU was designed and applied for graphics rendering purposes only. The processing units, traditionally called shaders, were initially fixed-functioned. The first programmable shaders were introduced in 2001[56]. This functionality triggered the development of shading programming languages. We note the Cg programming language[58], the OpenGL shading language (also known as GLSL)[38] and Microsoft's DirectX High-Level Shader Language (i.e. HLSL). The emergence of these languages enabled the programmability of the processing units within the GPU not only for graphics rendering but for solving general purpose problems which could naturally map to the graphics pipeline. In this way the GPGPU (General Purpose on Graphics Processing Unit) term was coined[50].

Research regarding the possibilities of GPGPU pushed the development of special programming languages for GPUs with a focus on GPGPU, e.g. Brook[16]. However, the actual breakthrough was set when the NVidia's Tesla architecture[55] as well as pure GPU compute programming languages were introduced. Tesla architecture employed unified shaders which no longer were designed for a particular purpose but they could either be used for vertex, fragment processing or computation. The first GPU adopting Tesla architecture was the GeForce 8800 GPU in 2006, which featured 128 streaming processors with a theoretical peak of 518 single precision GFLOPS performance. Pure GPU computation languages emerged, e.g. CUDA[73] and OpenCL[39]. These languages enabled the programmability of GPU for general purpose computation without involving the graphics pipeline. In this regard, a great interest from the research community was spawned in the adoption of GPUs for general algorithm acceleration.

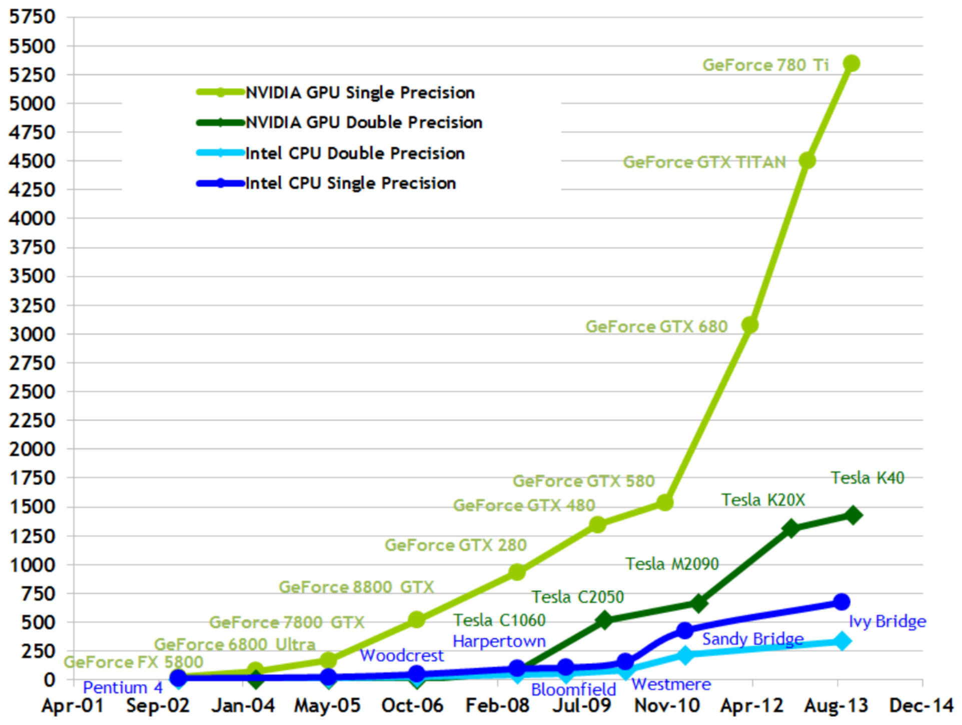


Figure 1.1: A comparison of CPU and GPU floating point performance evolution [73].

The GPU performance has vastly improved over the last years (figure 1.1). In 2010 the highest performing NVidia GPU (GTX-580) featured a theoretical peak of 1,581 GFLOPS single precision, whereas in 2015 the highest performing GPU of the same vendor (Titan-X) was rated at 6,144 GFLOPS, which is a 3.9x improvement. At the time of writing of this paper the highest performing GPU to date was the NVidia GP100. According to [76] the features of this GPU compared to the professional GPUs of the 2 previous generations in table 1.1. Half precision has been pushed mostly with the artificial intelligence applications in mind, e.g. deep learning methods. This performance plus the accompanying energy efficiency of the GPU are the greatest incentives for the adoption of the GPU for general purpose computation.

CUDA programming environment[73] has been developed by NVidia and it works exclusively on NVidia hardware. It is supported on all three major operating systems and it is the most established GPU computing programming environment to date. It is well supported by the vendor and most research on GPUs has been conducted by leveraging this particular API (Application Programming Interface). The most significant drawback of CUDA is the restricted hardware support, which limits users to the particular vendor's hardware.

The OpenCL is an open standard that targets multiple vendors and it is supported by the Khronos group consortium. Programming style is similar to CUDA though more verbose. However, it's not established to the same extent as CUDA though it receives steadily more attention. One of the primary advantages of OpenCL is the wide type of hardware support as it is not limited to GPUs but it can be applied on CPUs, DSPs, or FPGAs, as well.

Table 1.1: Tesla P100 (GP100) GPU compared to prior GPU generations[76].

Tesla Products	Tesla K40	Tesla M40	Tesla P100
GPU	GK110 (Kepler)	GM200 (Maxwell)	GP100 (Pascal)
SMs	15	24	56
TPCs	15	24	28
FP32 CUDA cores / SM	192	128	64
FP32 CUDA cores / GPU	2,880	3,072	3,584
FP64 CUDA cores / SM	64	4	32
FP64 CUDA cores / GPU	960	96	1,792
Base clock	745 MHz	948 MHz	1,328 MHz
GPU boost clock	810/875 MHz	1,114 MHz	1,480 MHz
Peak FP32 GFLOP	5,040	6,840	10,600
Peak FP64 GFLOP	1,680	210	5,300
Texture units	240	192	224
Memory interface	384-bit GDDR5	384-bit GDDR5	4,096-bit HBM2
Memory size	Up to 12 GB	Up to 24 GB	16 GB
L2 cache size	1,536 KB	3,072 KB	4,096 KB
Register file size / SM	256 KB	256 KB	256 KB
Register file size / GPU	3,840 KB	6,144 KB	14,336 KB
TDP	235 Watts	250 Watts	300 Watts
Transistors	$7.1 \cdot 10^9$	$8 \cdot 10^9$	$15.3 \cdot 10^9$
GPU die size	551 mm^2	601 mm^2	610 mm^2
Manufacturing process	28-nm	28-nm	16-nm FinFET

Currently, GPUs have mostly found a place on the HPC sector as the interest of the scientific community is significant. This interest stems from the vast compute requirements of scientific applications which tend to require enormous amounts of computations with inherent parallelism, which pose the GPUs suitable for such kind of algorithms. Mainstream use of GPU compute is still limited mostly to video and image processing, or brute force workloads (e.g. cryptocurrency mining, cryptography). Due to the architectural differences of CPUs and GPUs, performance optimization decisions have to be applied in a different manner in order to attain high performance. Since, the goal of GPU computing paradigm is reaching higher performance levels the proper use of optimization techniques is mandatory.

1.2 The performance analysis challenge

As already noted, the real focus of GPU computing is about performance so it would be of great significance to be able to predict performance of GPU applications on a wide range of hardware. Performance modeling information is particularly important that can be exploited for either the consideration of a hardware upgrade or even on choosing important optimization decisions. Additionally, the maximum performance that could be achieved is very important as this could aid in searching for any worthwhile optimizations. However, performance impact of migrating to a GPU accelerator or moving from one type of GPU to another can be a puzzling process to predict. Performance bottlenecks can be different due to architectural changes or variations on the balance of processor resources between different types of processors.

CPUs do not require a vast amount of parallelism in order to yield decent performance. They utilize large hierarchies of cache memories that are able to alleviate the large access latencies of main memory. In addition, they employ advanced techniques in order to maximize the single threaded performance, e.g. aggressive speculative execution, register renaming, result value forwarding, etc. All these features eliminate pipeline and memory bottlenecks, leading to more predictable execution results.

On the other hand, GPUs are significantly more performance sensitive to supplied parallelism, resource usage and memory access patterns. They are considered as massively parallel compute devices as they practically need many thousands of active threads in order to keep them occupied. This fact poses large problems with abundant parallelism as a requirement. The GPUs feature much smaller cache memories which in conjunction with the large amount of active threads allows only limited use, mostly for exploiting the spatial locality between sibling threads. The missing of large cache hierarchies forces programmers to effectively use memory. However, GPUs require regular memory accesses with specific requirements in order to apply coalescing, which is a mandatory requirement for efficient memory accessing. All reasons above induce potential bottlenecks for GPU performance. Practical experience has proven that GPU performance is sensitive to design decisions and fine tuning. In general, GPUs tend to be less tolerant to naive programming practices in regard to performance. Moreover, though GPUs provide great compute performance, this can only be achieved on problems that match their characteristics.

It should be added that GPU ISAs (Instruction Set Architectures) are not fixed, which can also affect the performance estimation process between different ISA GPUs. Modern CPUs tend to be compatible on a specific ISA mostly due to the necessity of software compatibility. The large existing software install base locks the CPU vendors on particular ISAs, though they implement completely different micro-architectures within their products. On GPUs such restriction does not exist as the software install base typically does not directly embed machine code but regularly binary code in other intermediate language forms. That said, typically GPUs execute different machine codes for the same kernels as they are just-in-time compiled for the respective ISA. Such differentiations can potentially change the used instruction mix significantly and subsequently cause performance variations.

For all the reasons above this thesis is focused on proposing a performance model that provides the necessary abstraction in order to be applicable on a wide range hardware, yet it provides decent prediction accuracy, is quick and straightforward to apply and can be fully automated based on *black-box* kernel inspection. In addition, this model was developed as roofline based and as such it is able to indicate an upper bound on performance, which can be fairly useful to the programmer as a guidance, providing performance feedback for further optimizations. The ultimate goal was to provide a tool that runs automatically the whole performance prediction process by utilizing an existing GPU program and producing the final results without the user's intervention.

1.3 Structure of thesis

This thesis is structured as follows. In the next chapter background information and a summary of related work is provided along with their connection to this thesis. The background information includes a short historical overview of the GPU as it evolved from fixed function graphics acceleration hardware to the point being considered as a special general purpose computation coprocessor. The various programming environments developed to date are also described. Related work includes relevant research work conducted in the field of performance prediction models for GPUs, both analytical and simulations.

The third chapter develops a discussion on the foundation of the proposed model, i.e. the roofline model, as well as, an introduction to the *quadrant-split* visual model. The roofline model is a widely accepted tool for performance analysis and the proposed model is based on the same concept. An alternative visual representation proposed is the *quadrant-split* model giving additional insight on multi-processor performance analysis. Furthermore, the roofline model was experimentally approximated by using an artificial micro-benchmark dubbed *mixbench*.

The fourth chapter is essentially the main contribution of this work. It provides a detailed description of all steps required by the proposed method along with an analysis of the proposed performance model. The description is additionally complemented with two applied case studies and validation with real performance measurement.

The fifth chapter is an extended experimental evaluation of the proposed performance method. Experimentation on a wide range of real world kernels is provided and a further analysis on special cases where the prediction was not considered adequately accurate. Additionally, the kernel performance of *mixbench* micro-benchmark was also validated by the exact same methodology.

The sixth chapter exhibits the potential of the proposed methodology in cross-vendor GPU environments (AMD GPUs) and a summary of the assumptions made in the performance model along with known limitations that have been identified.

The last chapter concludes by summarizing on the proposed performance model and initiating a discussion on possible future improvements. The latter includes potential improvements on the model itself, on the hardware support or on software assisted approaches.

2. BACKGROUND AND RELATED WORK

2.1 The graphics accelerator as a Graphics Processing Unit (GPU)

Initially, the graphics accelerator had emerged as an ASIC (Application-Specific Integrated Circuit) that performed fixed functions involved in 3D graphics rendering. In fact, well before 3D hardware acceleration was established there had been graphics acceleration hardware that focused on plain GUI 2D raster operations. The purpose of 3D hardware acceleration was the offloading of these tasks from the CPU to another device with the additional speed-up benefit. A simplified view of the graphics pipeline is illustrated in figure 2.1. Graphics accelerators gradually evolved by taking on the whole graphics pipeline.

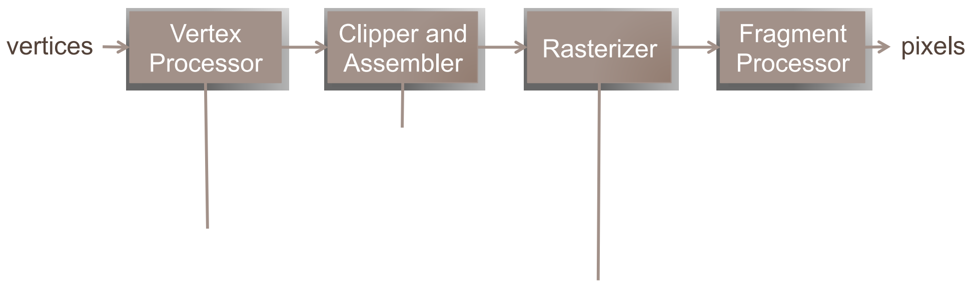


Figure 2.1: The graphics pipeline.

The term GPU (Graphics Processing Unit) was first coined by NVidia for the purpose of marketing of the newly launched GeForce 256 accelerator in 1999. According to the original definition a GPU was presented as a *”single-chip processor with integrated transform, lighting, triangle setup/clipping, and rendering engines that is capable of processing a minimum of 10 million polygons per second”*[1]. This term has been established for all modern graphics accelerators ever since.

Initially, no programming could be applied on the graphics pipeline as all functions of the GPU were fixed. However, as the accelerators evolved they allowed programmers do write small custom programs that could be applied to the data processed within the pipeline. These programs are called *programmable shaders*. The first GPU that offered this capability was the GeForce 3 (NV20). It was introduced in 2001 and added programmable pixel and vertex shaders. Thus, programmers could write programmable shaders involved in either the vertex processor stage or the fragment processor stage. This capability allowed the implementation of custom visual effects in graphics applications. Furthermore, it was the feature that enabled the capability of using GPUs for a broader set of problems beyond graphics.

[1]`http://www.nvidia.com/object/gpu.html`

2.2 The GPU as a general purpose processor

CPUs have traditionally been the heart of the computer as it is the primary component that manages and executes the program's instructions. All programs consist of a series of machine code instructions and each instruction describes a tiny step in the progress of computation. The solution of computational problems typically require a vast amount of instructions to be executed in order to reach the final output of a program. Therefore one of the primary factors in the speed of execution of a program is the speed of execution of instructions as permitted by the central processor, i.e. the CPU. Especially the HPC sector employs algorithms that require enormous amounts of computations to be performed. Therefore, the effectiveness of CPUs in the execution of computation is of primary focus in the particular industry.

During last decades the evolution of CPU has been mostly driven by the stable and fast-paced evolution of process technologies employed by the semi-conductor industry. This allowed the longevity of the so called Moore's law[62] which had been first observed by Gordon Moore, one of the founders of Intel corporation. This law was an observation which described a doubling every year in the number of components per integrated circuit and latter it was revised to doubling every two years. The increasing trend in the number of transistors on the CPU combined with increasing clock frequencies allowed the exponential scaling in the performance of CPUs which lasted for many decades. This improvement had been enjoyed by the software industry without applying any changes to the existing software. However, that era has reached to end due to the *power wall*[79] that seized the long term increasing of CPU frequency clock speeds and the diminishing improvements in exploiting the instruction level parallelism in existing software. Then, CPU vendors turned to multi-core designs that allowed the inclusion of multiple processors within the same chip silicon. IBM Power 4 was the first CPU to employ a multi-core design on year 2001. However, this move forced software designers to change their sequential approach in program development to parallel in order to take advantage of this new trend. An expressive quote used to present this new reality was *'The free lunch is over'*[87, 88].

On the other hand, the graphics accelerator employed a parallel design since the early beginnings. The parallel nature of graphics processing allows a huge number of operations to be performed in parallel and thus, GPU vendors adopted widely parallel processing designs on GPUs. For instance, even the GeForce 256, introduced in 1999, employed quad pixel pipelines. Thereafter, the additional transistors per fabrication node generation led the GPU performance to rapidly increase.

Initially however, the GPU didn't offer the programmability required to exploit this compute capacity for anything more than graphics rendering. Later on the programmable shader capability was introduced in 2001, as already reported, but the hardware units (shader units) that were designed to execute shader code were still specialized to the particular unit type. This meant that there were different physical units on the GPU at the time for each pipeline stage, i.e. vertex shader units and pixel/fragment shader units, each built for the execution of particular shader type codes. This led to imbalances in execution loads inducing negative performance impact.

2.2.1 Unified shaders through the introduction of the Tesla GPU architecture

The real breakthrough in the GPU computing field was set when the first desktop GPU with unified shaders was released. That was the NVidia GeForce 8800, which was introduced in 2006 and employed the Tesla GPU architecture[55]. It enabled flexible programmability through CUDA[73], a native GPU compute programming environment.

Some key information of the Tesla architecture[55] (figure 2.2):

- 128 streaming-processor (SP) cores organized as 16 streaming multiprocessors (SMs) in eight independent processing units called texture/processor clusters (TPCs)
- The SM manages and executes up to 768 concurrent threads in hardware with zero scheduling overhead.
- The SM is a unified graphics and computing multiprocessor that executes vertex, geometry, and pixel-fragment shader programs and parallel computing programs.
- The SM consists of eight streaming processor (SP) cores, two special function units (SFUs), a multithreaded instruction fetch and issue unit (MT Issue), an instruction cache, a read-only constant cache, and a 16-Kbyte read/write shared memory.
- The GeForce 8800 Ultra clocks the SPs and SFU units at 1.5 GHz, for a peak of 36 GFLOPS per SM. To optimize power and area efficiency, some SM non-data-path units operate at half the SP clock rate.
- To efficiently execute hundreds of threads in parallel while running several different programs, the SM is hardware multithreaded. It manages and executes up to 768 concurrent threads in hardware with zero scheduling overhead.

In order to support a large number of concurrent hardware threads, the SM sustains a large set or registers (8,192 total 32bit registers per SM) in order to be able to perform zero-cost switching whenever required. This allows supporting very fine-grained parallelism in an efficient manner. The SM features a flexible type of SIMD (Single-Instruction, Multiple-Data) architecture which designers call *single-instruction, multiple-thread* (SIMT). The SM's instruction unit creates, manages, schedules, and executes threads in groups of 32 parallel threads called warps. In essence, each warp maps to 32-lane SIMD operations and is able to follow an independent code path. During the execution, individual threads can be inactive due to independent branching. The SM maps the warp threads to the SP cores, and each thread executes independently with its own register state. The SM executes instructions at full efficiency and performance when all 32 threads of a warp take the same execution path. In case the threads within a warp diverge due to a data dependent conditional branch, the warp serially executes each branch path taken, disabling threads that are not on that path. After the execution of all taken paths complete, the threads re-converge on a common execution path. Different warps can freely follow independent execution paths without incurring negative performance impact.

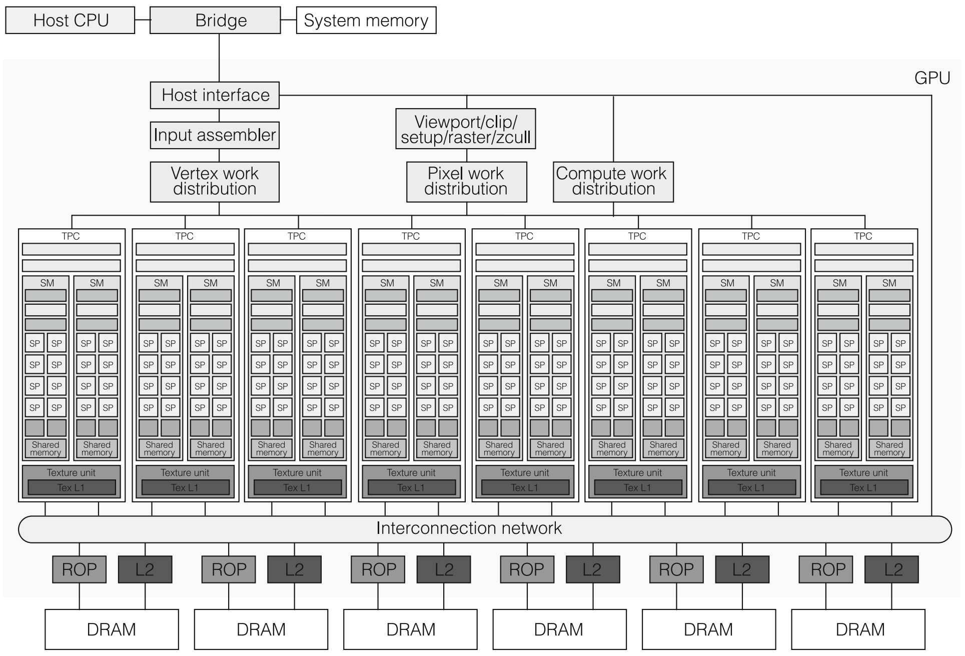


Figure 2.2: The NVidia Tesla GPU architecture[55] block diagram.

SIMT execution fashion is similar to SIMD. The difference is that SIMT applies instructions to multiple threads instead of multiple data lanes. In essence, the SIMD architecture and its vector width is not directly exposed the GPU ISA architecture. The GPU architecture instructions are scalar but they are executed by multiple threads in lockstep. It is a duty of the hardware to manage the SIMD execution. For that reason, this architecture allows programmers to ignore the SIMT execution implications and write parallel code in a MIMD fashion, by letting the hardware execute coordinated threads in a SIMD fashion. However, they can optimize performance by minimizing the circumstances where thread warp execution diverges. In contrast, typical SIMD vector execution requires from the programmer to explicitly handle vector operations and manage divergence, including vector loads and stores from memory.

The SM warp scheduler operates at half of the SP's clock rate. At each cycle, it selects one warp which is ready for execution to issue an instruction. The selected warp executes the issued instruction over four SP cycles. The warp scheduler issues instructions to the SPs and SFU units on alternate cycles, potentially keeping both fully occupied. These specifications enable the GeForce 8800 Ultra GPU a theoretical peak performance of 36×16=576 GFLOPS.

The supported ISA includes floating point, integer, flow control, memory load/store and texture operations. The integer and floating point operations include add, multiply, multiply-add, minimum, maximum, compare, transcendental, bitwise and conversions operations. Memory access supports read/write accesses on the three distinct address spaces, i.e. local memory, shared memory and global memory. In addition there is a constant memory address space offering read only access capability and a texture unit which allows cached read-only accesses in addition to sample bilinear filtering. Multiple global memory accesses can be coalesced, as long as they are aligned and reside on the same memory segments, one fact that can optimize use of memory bandwidth.

The Tesla scalable parallel computing architecture enabled the GPU to be effectively adopted for high-performance computing applications. Data parallel design embraces a two level decomposition, where the problem is first decomposed to blocks and then, the blocks are further decomposed to small problems that can be executed in parallel. This design maps naturally to the architecture of the GPU. The blocks map to the SMs and they are executed without any synchronization among them. The smallest tasks are then mapped to CUDA threads of CTAs (Cooperative Thread Arrays) which are threads belonging to groups that have the ability to synchronize and exchange data in an efficient manner. This kind of two level parallelism allows great scalability as the blocks can be distributed to as many GPU resources are available.

The fundamental features set on this GPU architecture have been carried over to Fermi architecture [63] and on the currently most recent GPU architectures. Some aspects have improved, though, but the primary architecture has stayed intact employing a similar two level SIMT thread hierarchy and fine grained scheduling.

2.2.2 A comparison of the GPU and the CPU

The CPU and the GPU have fundamental differences from the perspective of general purpose computing. CPUs are traditionally optimized on sequential programs with a focus on the minimization of latencies. GPUs, in contrast, focus on maximizing throughput by relaxing the latency restrictions and pushing parallelism to extreme levels. These natural differences reflect to many design aspects including the threading capabilities, core datapath, as well as, the memory hierarchy.

As already described, GPUs do excessive use of fine grained multithreading. CPUs on the other hand, rely on coarse grained multithreading in order to scale performance by duplicating the CPU resources and to increase its utilization by employing symmetric multithreading. However, multithreading is limited on CPUs. A typical 4 core desktop CPU provides a total of 8 hardware threads at most. A recent NVidia GPU allows a total of up to 64 active warps of threads, per SM. The GPU employs excessive multithreading in order to hide pipeline and memory access latencies.

A fundamental difference between SIMD, as it is applied on the CPU, and SIMT execution lies in flexibility. CPU SIMD instructions are not always applicable as their use is dependent on the access patterns and conditional operations. In addition, programmer has always to have the width of SIMD in mind. On the other hand, SIMT execution offers a great programmability advantage as every type of instruction is able to execute in a SIMT fashion and the conditional branches can be suitably handled by the hardware. The only implication of conditionals is their negative impact on performance, as they do not affect the implementation.

At the time of writing of this thesis, contemporary desktop CPUs were equipped with up to 10 cores. The largest consumer GPUs featured more than 20 SMs. In addition, one should consider the parallelism per core, per SM. A GPU SM can regularly perform a count of operations to the order of hundreds per clock whereas a CPU performs to the order of tens. Therefore, the overall parallelism characteristics of the GPU are far more evident compared to the CPU.

A modern CPU is equipped with 3 to 4 levels of cache memory. Typically, the L1 cache is exclusive to each core whereas L3 & L4 are shared among cores. On the other hand, GPUs feature a simpler cache memory structure, typically organized in a 2 level hierarchy (L1 & L2). The L1 is dedicated to each SM whereas L2 is shared. GPU caches in general are optimized for spacial locality instead of temporal locality. A significant cause for the statement above is the necessity of servicing a huge number of active threads, the memory accesses of which will largely evict any cached data that is intended to be reused. This effect eliminates any opportunities to exploit temporal locality. For this reason, the GPUs are typically suited for applications that employ specific patterns on data reuse.

On the other hand, the GPU main memory provides higher memory bandwidth compared to the CPU. The GPU memories are optimized for bandwidth, though, their access latencies are significantly higher. Typical bandwidth for GPUs is rated to hundreds of GB/sec which is an order of magnitude higher than the typical bandwidth of a CPU.

Additionally, each GPU SM has far more available register space than a CPU core does. The large register files are required in order for the GPU to keep the context of vast amounts of active threads on the SM, enabling zero cost thread scheduling. As mentioned earlier, each SM is able to support thousands of active threads. In order for this to be this possible, the register file must be sized to hundreds of KBytes.

Comparison of particular CPU and GPU products

As a more direct comparison, the GP100 GPU[76], the specification of which has already been provided in table 1.1, is compared with an equivalent high-end server class CPU, the Intel Xeon Intel Xeon Processor E7-8890 v4. Both products are HPC oriented and used on servers. A direct comparison of quantitative features that were discussed previously is presented in table 2.1.

The particular CPU consists of 24 CPU cores whereas the GPU is equipped with 56 SMs. Each CPU core is able to keep just 2 active hardware threads through hyper-threading technology. In contrast, each SM is able to keep a maximum of 64 warps active. This entails a huge difference on the maximum hardware threads per processor ($\approx \times 75$).

Similarly, the register file per SM on the GPU is more than $\times$21 the register file of the CPU. The recent Intel Skylake architecture designates the CPU cores having 180 physical integer registers plus 168 physical floating point registers. This is a total of $180 \times 8 + 168 \times 64 = 12{,}192$ bytes per core, assuming the use of AVX512 SIMD extensions (512bits per floating point register and 64bits per integer register). On the contrary, each SM of the NVidia P100 GPU is equipped with a 256KB register file, which is far larger.

Table 2.1: A comparison of Xeon E7-8890 v4 CPU and NVidia GP100 GPU features[76].

Feature	NVidia GP100	Intel Xeon E7-8890 v4	Ratio
Core/SM count	56	24	2.333
Flops per clock, per core/SM	64 DP Flops	16 DP Flops	4.000
Total hardware threads/warps	3,584	48	74.667
Total last level cache	4 MB L2	60 MB L3	0.067
Total per core dedicated cache	3,584	6,912	0.519
Register file size per core/SM	262,144 bytes	12,192 bytes	21.501
Total register file space	14,336 KB	285.75 KB	50.170
Memory bandwidth	720 GB/sec	85 GB/s	8.471

On the other hand, cache memories are dramatically smaller on the GPU. Although, the GPU has 4MB of L2 cache, which is the largest recorded to date for a GPU, the CPU features a $\times$15 larger L3 cache. Furthermore, the total core dedicated cache is almost double on the CPU ($24 \times 256 + 768 = 6{,}144 + 768 = 6{,}912$ vs $56 \times 64 = 3{,}584$). However, the small memory cache is alleviated by the exceptionally high memory bandwidth, which is more than $\times$8 on the GPU and it is provided by the stacked HBM2 memories.

The harnessing of parallelism was evident on the GPUs since the early beginnings. Since, this trend only became more aggressive and at the time of writing of this thesis the high-end GPUs featured well over 3,000 parallel ALUs (Arithmetic Logic Units), e.g. the NVidia GP100[76] and the AMD Fiji with 3,584 and 4,096 shader units, respectively. This aggressive parallel architecture of GPUs pushed them to extreme levels of compute capabilities. As already presented in section 1.1 the GPU compute performance evolution has been greatly increased over the last years. This is mostly attributed on the fact that the graphics rendering process offers huge parallel floating point computation opportunities and the GPU designs were built to exploit this abundant parallelism. The clock frequency wall did not have a severe negative impact for GPUs whereas CPUs suffer to exploit additional instruction level parallelism from existing codes.

2.3 GPU compute programming paradigms

Early adoption of GPUs for computation employed shader programming languages though the available graphics APIs. Some popular shading languages at the time were the OpenGL shading language (also known as GLSL)[38], the NVidia's Cg programming language [58] and Microsoft's DirectX High-Level Shader Language (i.e. HLSL). These paradigms forced researchers to map their computation problem into a graphics problem. Therefore, it required a knowledge background and experience on the graphics APIs and pipelines.

GPUs intrigued the interest of researchers since the early beginnings. Some indicative early work on GPUs for general purpose computation is the work of Dokken *et al.* describing methods and challenges of GPGPU, along with an example of solving the heat equation using a Jacobi iteration [25]. An additional implementation of the Jacobi iteration computation was proposed by Amorim *et al.* [83], who presented an implementation using both CUDA and OpenGL shaders (GLSL). Another example is an intrusion detection system by Jacob *et al.* [34] which offloads string-matching computations to the GPU and it is implemented in Cg.

In order to alleviate the programmer's involvement with graphics management, the Brook research project[16] was developed as a special GPU compute language that could hide the graphics peculiarities from the programmer. However, the particular language, though being a promising project, never did gain wide acceptance.

The CUDA environment[64, 73] was the first native GPU compute development environment which consisted of a complete toolchain for GPU compute programming and was independent to the already existing graphics APIs. It was first introduced in 2007 by NVidia and since, it has been the most popular GPU computing toolkit available. It supports all modern NVidia GPUs, commercial and professional, and the most common operating systems, i.e. Windows, macOS and Linux. It provides a rich API on the host processor for the GPU management and a C/C++ like kernel language for GPU kernel programming.

One of the most significant disadvantages of CUDA is its hardware support limitation to NVidia GPU hardware. This gap is filled by an alternative GPU computing library called OpenCL (Open Compute Language) and directed by Khronos group[39]. OpenCL is an open standard which is supported by most hardware vendors. Its purpose is not limited on GPUs, but it has been also supported by CPUs, DSPs and FPGAs. It has the form of a library and it accepts kernels in text form through function calls which are compiled by the device driver in a JIT (Just In Time compilation) fashion. The OpenCL programming model is quite similar to CUDA. The kernel language is a C derived language called OpenCL C. Beyond CUDA, OpenCL is one of the most popular GPU programming paradigms.

CUDA and OpenCL are considered as low level GPU programming models as they expose significant details of GPU execution and provide almost full control of the device. As such, the programmer needs to have a deep knowledge of the GPU hardware in order to achieve acceptable performance. In addition, a problem raised by using low level programming models is the significant time required for the development of optimized code which is done at the cost of productivity. This need has led to higher level GPU programming models which facilitate productivity.

OpenACC[77] is one of the most well known high level programming languages for GPU compute. It is directive based and it allows the same source to be compiled for CPU if it is not supported by the compiler. In this sense, it significantly resembles the OpenMP[23] parallel programming standard, though it has been developed specifically for GPUs.

Similarly, the OpenMP standard[23, 78] has expanded its support to accelerators since its version 4.0. It is directive based, similarly to OpenACC. However, it is a well established programming paradigm for shared memory multi-processors and it has only recently been extended support to GPU accelerators.

Another higher level programming model for GPUs is C++AMP[60]. C++AMP is a standard specification defining a set of extensions on top of the C++ language on the purpose of GPU compute acceleration. It has been introduced by Microsoft corporation and the implementation they developed leverages the DirectX API as a backend, restricting the OS platform to Windows. Recently, however, a C++AMP implementation has been introduced by AMD through their open source HCC compiler[84] which leverages the LLVM compiler infrastructure[49] and is available for Linux OS. This implementation uses the ROCm[7] software stack as a backend, which is an open source software stack specifically developed for professional GPU compute.

Finally, apart from programming languages for GPUs the diversity of GPU architectures imposes the use of intermediate languages in order to split the compilation process to more and simpler stages. Typically, each one of the GPU compute platform implementations defines its own intermediate language. As a reference we provide the most common in table 2.2 along with the respective platform.

Table 2.2: Most common intermediate languages leveraged by GPU computing platforms.

Intermediate Language	GPU computing platform
PTX (Parallel Thread eXecution)[74]	NVidia CUDA
SPIR/SPIR-V (Standard Portable Intermediate Representation)[35]	Khronos OpenCL
HSAIL (HSA Intermediate Language)[33]	HSA Runtime
AMD IL (AMD Intermediate Language)[3]	AMD OpenCL runtime

2.4 GPU performance modeling

The GPUs have offered additional opportunities in the high performance computing field by being utilized as accelerators in dense computations. Since they play a key role in performance computing and their architecture is significantly different from the more traditional processors, i.e. CPUs, they have triggered the researchers' interest in designing performance models.

GPUs, as opposed to CPUs, are throughput processors. They mostly have different performance bottlenecks and peculiarities than CPUs due to their different characteristics. The different amounts of caches, the vast concurrency, increased latencies, higher compute throughput and memory bandwidth are some factors. A performance model should, at least, focus on the performance critical factors of the GPU. The different characteristics of the GPU impose the use of tailored performance models on this special type of hardware.

The work that has been developed to this goal can be classified into two major categories. First is the simulation approach which can be extremely accurate yet a time consuming process. The analytical approaches are much faster and easier to apply. Furthermore, they tend to use and provide a better high-level insight [30]. Simulation based predictions may use vast amount of generated data that may not yield the desired insight that is easy to be interpreted by GPU programmers. Using an analytical performance model with a reasonably small number of parameters could lead to a model easy to use by programmers while producing predictions that are sufficiently accurate.

In the following paragraphs simulation based modeling approaches are first presented and next the analytical approaches to performance estimation follow. Additionally, some more special case approaches are presented. Finally, power efficiency oriented, as well as, GPU micro-benchmarking research is described. The referred research work mostly employs the NVidia CUDA programming environment, unless otherwise stated.

2.4.1 Simulation based models

Here some significant related work on GPU simulators is presented. Bakhoda *et al.* developed a simulator called GPGPU-Sim[9], which simulates the execution of PTX instructions. The authors employed the simulator to characterize the performance impact of several micro-architecture features, including the interconnect topology, the use of caches, the design of memory controller, the parallel workload distribution mechanisms and the memory request coalescing hardware. In their experiments they observed that performance is more sensitive to the interconnect bisection bandwidth rather than latency and that, for some applications, running fewer threads concurrently than the GPU allows can improve performance by reducing contention in the memory system. At the end of this section more detailed information is provided regarding the GPGPU-Sim simulator.

Power *et al.* developed an heterogeneous CPU-GPU simulator called gem5-gpu[80]. It is based on the aforementioned GPGPU-Sim simulator [9] and on the gem5 CPU simulator [12]. It is able to support coherent caches and a single virtual address space with separate physical address spaces on CPU and GPU.

Multi2Sim simulator [89] is yet another open source CPU and GPU simulator. In [89] authors used it to simulate an x86 CPU and an AMD Evergreen GPU. It has been extended though to support the more recent AMD Southern Islands GPU architecture.

Another PTX execution simulator is Ocelot, which has been developed by Kerr *et al.* [37]. Ocelot can execute compiled kernels from the CUDA compiler and it supported the full PTX 1.4 specification at the time authors had written their research paper. The authors used the simulator to provide experimental results that quantified the impact of optimizations, e.g. branch re-convergence, data sharing between threads and additional parallelism.

A special software tool that enables simulated execution of GPU programs is Oclgrind [81]. The particular software allows running OpenCL applications on a virtual OpenCL software device. In this sense, it cannot be strictly considered as a GPU simulator, yet it allows GPU program execution analysis and bug detection through software. More information on the tool follows.

Most tools discussed in this section are provided under open source licenses and therefore are freely available for experimentation. Two representative tools are further described here.

GPGPU-Sim

The GPGPU-Sim simulator [9] has been extended to support CUDA and it incorporates all essential architectural features of a GPU. It supports the notion of a shader core which is equivalent to the Streaming Multiprocessor and each core consists of an SIMD unit, the width of which is the same as the number of SPs per SM. The cores are connected to a set of memory controllers through an interconnection network and each controller features an L2 cache. Each shader core consists of a set of fast memories, i.e. L1 texture cache, L1 constant cache, L1 local & global caches and shared memory. The simulator supports essential features of GPU execution including multi-stage pipelines, thread warp scheduling in groups of 32 threads, memory access transaction coalescing and a round robin scheduling of warps supporting long latency operations (e.g. DRAM accesses). Essentially, the simulator accepts PTX kernel code as input for execution.

The initial work [9] was focused on a GeForce 200 based GPU model, thus the width of the SIMD had been set to 8. At the time of writing of this thesis, though, the simulator had been extended with support of the Fermi GPU architecture. It provides configuration with a wide range of architectural options. It is believed that it can be extended in a straightforward way to support even more recent GPU architectures.

Oclgrind

Oclgrind is a tool that acts as a virtual OpenCL device on the system[81]. It enables the analysis and debugging of OpenCL programs. It features a plugin interface providing extensibility through which sophisticated developer plugins can be developed for the collection of individual execution metrics in order to satisfy complex analysis.

The tool executes kernels in an architecture-agnostic manner. No actual execution on a physical GPU is conducted as OpenCL kernels are assigned to the CPU for execution. The tool does not simulate any particular GPU features as it aims at correct semantical execution of OpenCL code conforming to the standard. The OpenCL kernel sources are compiled to SPIR (Standard Portable Intermediate Representation) code. Thereafter, SPIR code is executed by interpreting SPIR instructions.

The tool incorporates a plugin that can generate histograms of the different types of instructions executed by the OpenCL kernel. It enables the developer to inspect the approximate instruction mix and the amount of memory that is read or written to each memory type. As the interpreted instructions are based on SPIR, they won't precisely match the GPU ISA instructions of any actual hardware, yet the ability to perform run-time analysis on their execution is still worthwhile for performance investigation.

In [81] it is reported that Codeplay, a software development company specializing in creating compilers and tools for heterogeneous architectures, has developed an extended implementation of Oclgrind that gathers data on memory operations. According to the report, offline analysis of this data allows developers to identify code that is generating excessive off-chip memory traffic or causes poorly coalesced memory accesses. This tool exhibits the extensibility of Oclgrind as a performance analysis tool.

2.4.2 Analytical approaches

An early work example was done by Liu *et al.* who proposed GPU performance prediction models[57] which classify a GPU application to one out of three categories, i.e. data-linear, data-constant and computation-dependent. They employ a set of factors that affect performance: the time to load data to RAM, read back results to CPU, load shader program to fragment processor, do overhead operations (framebuffer initialization and texture binding and mapping) and perform the actual computation on fragment processor. As a pioneering work the implementations were developed in a graphics API, i.e. GLSL.

A high level abstract model was developed by Ma *et al.* who proposed a GPU performance theoretical model[48] called called "Threaded Many-core Memory". It is regarded as an improvement of the *PRAM* model[27]. This model extends on the *PRAM* model by supporting features as highly threaded execution and memory access latency hiding. Authors analyze the accuracy of *PRAM* and *TMM* on four algorithms for the classic all pairs shortest paths problem where they prove the superiority of the latter.

An analytical GPU performance model that is based on the occupancy of the memory subsystem and exhibits the memory warp level parallelism was proposed by Hong *et al.* [31]. This model considers the amount of parallel memory requests filling the memory pipeline by considering the amount of running threads, operational intensity and memory bandwidth. Authors employed the CUDA programming environment and they applied their model on micro-benchmarks and applications where they exhibited 5.4% and 13.3% geometric mean of absolute errors, respectively.

Another performance prediction model was proposed by Kothapalli *et al.* [47] on which they took into account of various special GPU characteristics, e.g. the amount of cores (SPs/SM), the effects of memory latencies, memory access conflicts, cost of computation, scheduling and pipelining. Their proposed model can be used to analyze CUDA kernel pseudo codes in order to obtain performance estimations. The authors performed experiments with matrix multiplication, list ranking and histogram generation kernels.

An analytical approach was also followed by Baghsorkhi *et al.* who performed performance prediction on GPU architectures[8]. The authors represent a GPU kernel as a work flow graph structure, which they analyze in order to estimate performance. They use benchmarks that stress different GPU micro-architecture events, e.g. uncoalesced memory accesses, shared memory bank conflicts and branch divergence, all of which pose challenging to analytical performance models. Authors validate their performance model on matrix multiplication and FFT kernels. It can be useful for either compiler applying optimizations or GPU programmers in order to assess performance bottlenecks in their code.

Stevens *et al.* in [86] propose a linear GPU performance model based on kernel parameters and GPU coefficients for numerical oriented kernels. They consider costs related with data motion, floating point computations and synchronizations. A set of benchmarks is used in order to fit the parameters to the characteristics of the applied GPU hardware. In this respect this model does not account for overlapping of compute and memory operations. The relative performance accuracy exhibited in their experiments is comparable to previously published work. Authors employed the OpenCL GPU programming environment and in this regard, their model can be used across GPU vendors and architectures.

Sim *et al.* proposed an analytical performance prediction framework with a focus on guiding the programmer to beneficial optimizations in order to improve performance[85]. The authors built a performance model which makes use of a wide set of parameters, e.g. detailed instruction counts, warp counts, and hardware parameters, e.g. SM counts, latency metrics, cache miss ratio, ILP (Instruction Level Parallelism), MLP (Memory Level Parallelism), SIMD width, etc. Their model is based on the MWP-CWP (Memory Warp Parallelism - Computation Warp Parallelism) model. It can generate programmer guiding metrics. The authors employ their proposed model on sample input codes in order to predict performance benefits.

In another pioneering work authors create a performance model based on the investigation of GPU ISA code and three major components, such as pipeline, shared memory and global memory access[94]. The authors employ micro-benchmarks on the assessment of GPU characteristics on the currently rather old GeForce 200 architecture. Their proposed model is able to identify GPU kernel bottlenecks and provide quantitative performance analysis allowing programmers to predict the benefits of potential program optimizations. The experiments include three GPU applications: dense matrix multiplication, tridiagonal systems solver and sparse matrix vector multiplication. They present experimental results with high accuracy (5–15% error) as the model exploits information based on the native GPU instruction set. In addition, authors employ their model to suggest architectural improvements on hardware.

Dao *et al.* in [24] propose two performance models for GPUs. A sampling based linear model and a more advanced machine learning model. The former cannot deal with coalescing or caching effects, whereas the latter can be used on either with or without caches. The authors employ the OpenCL API.

Other researchers have used statistical methods for the purpose of performance prediction on GPUs. Velho *et al.* employ a simple linear regression model for the purpose of performance modeling[91]. Their proposed model involves three components: dispatch time, execution time and collection time. They apply it on three common problems: matrix multiplication, FFT and NeedleMan-Wunsch algorithm. Ali Karami *et al.* propose a performance analyzer framework[36] focusing on OpenCL kernels which is based on principal component analysis (PCA) and a multiple regression model, utilizing data extracted by hardware profiling metrics. A multiple linear regression model was also employed by Mirsoleimani *et al.* in order to construct a performance predictor for GPUs[61]. Zhong *et al.* incorporated a performance model[95] on a tool they developed which aims to improve the utilization of GPUs by optimizing the scheduling of workloads through dynamic slicing along with concurrent kernel execution. The performance model they propose is based on Markov chain theory as they attempt to capture the non-determinism of concurrently executed kernels.

Authors in [54] utilize a performance model in their work involving a compute/memory bound analysis. It is based on IPC metrics of various operations and the maximum memory bandwidth of the GPU, combined with a set of kernel's instruction count parameters. Their primary focus is an algorithm for scheduling concurrent kernels, focusing on the optimized order of execution through combining kernels with opposing resource requirements, i.e. compute intensive along with memory bandwidth intensive kernels. They define the degree of the relative synergy among kernels as a measure for rating the extent to which kernels are symbiotic. Through symbiotic scheduling the authors exhibited both performance, as well as, energy efficiency gains.

Part of research work is focused on the analysis of specific aspects and resources which can be performance critical during GPU execution. Boyer *et al.* built the Grophecy++ framework which they use to predict speedups of GPU kernels with a particular focus on the CPU-GPU data transfer cost[15], the cost of which is a potential bottleneck when data is not resident in the device memory. The Grophecy++ framework extends on Grophecy [59], an earlier presented framework which is able to project the overall speedup from CPU code. In particular, Meng *et al.* in [59] describe a method for developing an abstract representation of a CPU code called skeleton, which they analyze by mapping its pieces to various code transformations that correspond to hypothetical GPU codes. Their framework does not require an actual GPU implementation. Later on, this representation can map to estimated performance as well as the cost of development of the particular GPU code, which can help developers determine whether GPU acceleration is beneficial before actual development is undertaken.

Van Werkhoven *et al.* are also focused on the CPU-GPU transfer cost and they developed a cost model that considers the PCIe transfers, as well as, the overlapping of communication with computation [90]. The authors explore the wide range of possibilities in the implementation of overlapping, e.g. the applied number of streams or the use of device-mapped host memory. They propose this model for the evaluation of all possible different implementations with respect to their performance.

Authors in [40] propose *CuMAPz*, a tool for analyzing the memory access patterns performed by GPU kernels in order to give guidance on optimizations. It focuses on analyzing the behavior of both shared and global memories. This is based on memory access simulation by considering many aspects, i.e. data reuse, global memory access coalescing, shared memory bank conflict, channel skew and branch divergence. The performance impact of these aspects is predicted on various ways using shared and global memories. The authors exhibited a 32% improvement on code over a previous approach.

A hybrid approach was employed by authors in [17], which is focused on exploiting the advantages of both simulation and analytical prediction approaches. The authors developed a tool designed for performance prediction, in which they employ both a simulator and an analytical model. They combine them in order to be able to do fast and accurate performance estimations.

The roofline model

The roofline model [92] introduced by Williams and Patterson, is a visual model that provides insight on the maximum expected performance of a kernel by considering both pure computation and DRAM memory transfer requirements. It is based on the assumption that performance is either bound on the compute potential or the memory bandwidth of the underlying processor. The performance bound is either one depending on the relative requirements of operations of the application. The compute to memory transfer performance ratio is expressed with O_{dev} as equation (2.1) shows, which is the fraction of the maximum compute operation throughput to memory bandwidth specifications of the device.

$$O_{dev} = \frac{Throughput_{(compute)}}{Bandwidth_{(memory)}} \tag{2.1}$$

Operational intensity is measured in *flop/byte* units and is used to determine the limiting performance factor on a particular processor. This can be applied by estimating the program's operational intensity which is determined by the respective program's requirements as equation (2.2) indicates:

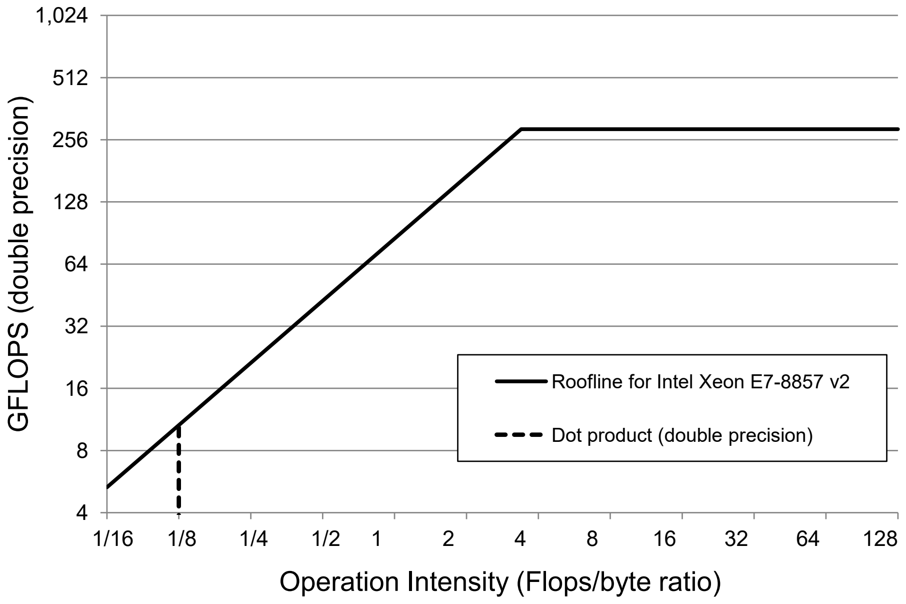


Figure 2.3: The roofline visual model for Intel Xeon E7-8857 v2.

$$O_{kernel} = \frac{Operations_{(compute)}}{Traffic_{(memory)}} \tag{2.2}$$

This fraction is also measured in *flop/byte* units but it is dependent on the application characteristics. Depending on the whether $O_{kernel} > O_{dev}$ the kernel is considered as compute bound or memory bound. The graphical representation of the roofline model is able to provide a quick and insightful visual representation of the device theoretical peak performance. In figure 2.3 the solid line represents the theoretical peak performance of an Intel Intel Xeon E7-8857 v2 CPU depending on the program's operational intensity. In this example for program operational intensities up to 3.39 flop/byte the program is considered as memory bound. Compute bound programs must exhibit higher compute intensity.

The expected theoretical peak performance of a device is estimated using equation (2.3):

$$Perf_{(roofline)} = \begin{cases} Throughput_{(compute)}, & \text{if } O_{kernel} > O_{dev} \\ O_{kernel} \times Bandwidth_{(memory)}, & \text{if } O_{kernel} \leq O_{dev} \end{cases} \tag{2.3}$$

For example, the computation of the dot product between two double precision arrays of length N requires traversing both of them. Thus, the expected memory traffic for accessing both arrays would be $2 \times N \times 8 = 16 \times N$ bytes. The amount of compute operations is $2 \times N + 1$, as each pair of values between the two arrays has to be multiplied and accumulated to the total sum. Therefore the operational intensity is estimated to be as follows (2.4):

$$O_{\text{dot product}} = \frac{(2 \times N + 1)}{16 \times N} \approx 1/8 \tag{2.4}$$

which in general is rather low. In the graphic representation (figure 2.3) one can check for the particular operational intensity which is the expected performance (intersection of dashed and solid lines). In this case, the Intel Xeon E7-8857 v2 features $O_{dev} = {}^{288}/_{85} = 3.388$ flop/byte, which is more than 25 times higher. Therefore, the roofline model points to a memory bound performance limitation for the dot product computation.

Therefore, using this principle can aid on the characterization of particular programs. Knowing the actual performance limitation is essential in order to follow proper decisions for optimizing a particular application on a specific hardware.

Roofline derived models

Higher level performance models are more hardware agnostic, consisting a hardware abstraction of the real device, but tend to be less accurate. Nugteren *et al.* proposed a high level performance model[65] which is roofline based, though refined by classifying the kernel under inspection to a specific type out of a set of predefined classes which are fine tuned instances of the roofline model, based on specific problem parameters. This way the performance model provides more accurate prediction results compared to a common roofline model approach. A significant highlight is that their proposed model inherently works without requiring an existing optimized code implementation, which turns the model ideal for estimating performance prior development.

Bombieri *et al.* designed a performance model[13] for identifying performance bottlenecks on GPU kernels, with the ultimate goal of taking optimization decisions. The authors identify the maximum potential performance speed-up by eliminating a particular bottleneck. They applied their model on three cases: Parallel Reduction, BFS and Matrix Transpose. Through the method they propose they showed the way their proposed model led to optimizations as a tool for providing guidelines to programmers.

Another roofline derived model is the work proposed by Li *et al.* in their “X model”[52]. It is based on an older proposed visual model by the same authors called “Transit” model and combines rooflines of memory bandwidth and compute throughput. Authors consider the ILP, TLP (Thread Level Parallelism), DLP (Data Level Parallelism) and MLP (Memory Level Parallelism) factors. Using the “X graph”, which is generated through the “X model”, GPU programmers can identify performance bottlenecks and evaluate the effectiveness of optimizations by investigating their individual and combined effects on performance.

2.4.3 Specialized application models

Other researchers have worked on performance models tailored for specific applications. Examples are the work of Guo *et al.* as well as Li *et al.* , who have focused on Sparse Matrix-Vector Multiplication (SpMV)[29, 53]. Guo’s goal is to accurately predict the kernel execution times of CSR, ELL, COO, and HYB SpMV kernels, without being limited by GPU programming languages or restricted to specific GPU architectures. The focus of Li is a matrix type independent method for performance analysis and optimization. Authors employ a probability mass function (PMF) method to optimize the SpMV computation parameters. Using this method they estimate the performance of SpMV based on COO, CSR, ELL, and HYB formats. They compare their proposed method to the proprietary cuSPARSE lib.

Baumeister’s *et al.* work regards to Finite-Difference Time-Domain (FDTD) applications[10], whereas Feichtinger *et al.* have built a performance model on lattice Boltzmann (LBM) simulations[26]. Baumeister focused on a particular implementation called B-CALM (Belgium-California Light Machine) and they adopted a simple, semi-empirical modeling approach to design a model which they validated for different hardware architectures. Feichtinger proposed their software framework called waLBerla, which is centered around the lattice Boltzmann method. They applied a highly scalable multi-GPU parallelization based on the well established MPI[28] standard in a hybrid parallelization approach capable of using CPUs and GPUs in parallel.

2.4.4 Power consumption oriented models

A measure gaining more importance recently is power consumption. This issue is even more significant given the fact that one of the major obstacle for High Performance Computing is power consumption. In this regard, researchers are also focusing on GPU power consumption and efficiency prediction models. For instance, Benedict *et al.* proposed a performance and power consumption model for GPU kernels[11] which is based on dynamic regression models and Dynamic Random Forests (DynRFM), Dynamic Support Vector Machines (DynSVM) and Dynamic Linear Regression Models (Dyn LRM) in particular.

Other research involving power consumption modeling on GPUs has been conducted by Abe *et al.* [1]. The authors present power and performance characterization models of GPU accelerated systems. They apply it on multiple architectures, including Kepler and Fermi, where they quantify the impact of voltage and frequency scaling for each one. Next, they propose statistical power and performance modeling of GPU-accelerated systems that is simple enough to be applicable for multiple GPU architectures. They argue that even though the proposed statistical models are simplified, they are able to predict power and performance of GPUs within reasonable errors (20% to 30%).

Wu *et al.* in [93] followed a machine learning approach. Their model employs a neural network that is trained by applying executions on real hardware and collecting the performance and power measurements. The model learns how the application's behavior scale as the GPU hardware configuration is changed by using the measured performance and power data. Performance counters of executions on kernels are used to feed the neural network in order to predict the scaling curve that represents a particular kernel. This curve is used to predict performance and power consumption of the application on different GPUs. The authors exhibited results with high accuracy which can be compared to GPU simulators. Moreover, after the initial training of the network the model is very fast on its execution.

Another power and performance model was proposed by Hong *et al.* [32]. Their primary goal is to find the optimal number of active processors for GPU kernels, based on the fact that if an application reaches its peak memory bandwidth then the exploitation of more cores is not expected to improve performance any further. The model they propose does predict both performance and power consumption. They exhibited they could save ≈ 11% of energy on average by using their method.

2.4.5 Micro-benchmarking of the GPU

Researchers often employ micro-benchmarking in order to dive into the unknown characteristics of compute devices and gain further understanding of the underlying hardware. The extracted information can be used for the purpose of performance modeling of the particular devices. Significant amount of research work has been based on benchmarking accelerators either for the purpose of designing a performance model or for the deeper understanding of hardware.

A benchmark similar to *mixbench* which is presented in this thesis was developed by Choi *et al.* who propose a model for the analysis of various hardware devices focusing on the execution time, power consumption and energy efficiency with respect to the operational intensity of the application under execution[19]. Their goal is to connect the operational intensity of the application with the efficiency of various types of devices used as building blocks in the HPC sector (GPUs, APUs, SOCs). The benchmark they developed is based on the same principles as *mixbench* does with some minor technical differences, e.g. the authors employed manual unrolling of loops, different handling of the results which are always written back to memory and each execution is designated for a specific operational intensity value, whereas *mixbench* is based on template variables with automatic unrolling, gets its intermediate results reduced but avoids writing them back to memory and invokes multiple kernels mapping to a wide range of operational intensities. For more information, reader is referred to paragraph 3.2 and section A.1 of the appendix, where the kernel's source code is provided.

Lemeire *et al.* [51] have also presented GPU micro-benchmarks for the analysis of issue and completion latencies in various operations. They conduct an analysis on the effects of the occupancy on these measurements, e.g. on a latency bound situation, or when the ILP is not enough. The OpenCL environment was chosen in order to allow cross vendor benchmarking.

The author of this thesis has also presented a set of micro-benchmarks for the evaluation of fast on-chip GPU memories[43]. A short discussion on this contribution is provided in the next paragraph, as well as, in B of the appendix.

2.4.6 This thesis primary contributions

The theoretical foundation of the work presented within this thesis has been presented in a preliminary stage as a regular conference paper [42] and subsequently this work was extended and published as an elaborate work in the form of a journal article [46]. The proposed performance model is considered as a high level approach based on the roofline model [92]. A discussion of the roofline model with GPU considerations in mind is provided in the 3rd chapter, along with a proposed visual representation called *quadrant-split* model, which can provide better insight. Finally, the detailed description of the proposed performance model is provided in the 4th chapter of this thesis and the experimental results follow in the 5th chapter.

The first paper contribution [42] presented an initial form of the method along with a limited number of experimental results. The relevant journal publication [46] extended the method to a fully automated prediction process. The experimental results included executions on a wide range of different real world kernels and a micro-benchmark. The hardware used for the experiments included 4 consumer and 2 professional GPUs. Furthermore, the proposed model was extended to the experimental use on a cross-vendor GPU environment by employing an AMD GPU and the exhibited results were quite promising.

Other contributions that have been used in this thesis include an implementation of a red-black SOR stencil computation method [44, 45] which has been utilized in the experiments and it poses as a proof of concept case study in this thesis. A theoretical performance analysis of the algorithm was provided and the implementations included various kernels, each utilizing a different memory caching approach. Additionally, a set of developed LM-SOR stencil computations [20, 21, 22] were also developed which served to investigate the re-computation strategy as an optimization. In this respect various implementations were investigated which are characterized by different operational intensities due to the different degree of re-computation applied. Implementations of this work were also applied on the performance model. Last, a set of micro-benchmarks [43] was presented that serves to the purpose of better understanding of the hardware capabilities regarding the GPU's fast on-chip memories. The micro-benchmarks assess the fast on-chip memories which include shared memory, L1 & L2 cache, texture cache and constant memory cache.

3. GPU ROOFLINE MODEL AND THE QUADRANT SPLIT REPRESENTATION

As already described on the previous chapter, the roofline model provides qualitative insight to the primary performance limiting factor given no additional latencies affecting performance. In this regard, a program can be classified as either compute bound when being limited by compute throughput or memory bound when being limited by memory bandwidth. Applying this principle on a GPU is a natural step as GPUs heavily rely on memory access latency hiding by overlapping computations [73]. GPUs support hardware context switching between threads that are active within a Streaming Multiprocessor. This context switching is practically cost free and GPUs have been designed to exploit thread level parallelism in this fashion in order to keep the resource utilization at high levels. This overlapping potential allows the clear classification of a GPU kernel as bound to compute or memory throughput, but not both. As such, it is a valid decision to apply the roofline model on GPUs, at least in the same sense as it has been applied on CPUs.

3.1 Roofline GPU considerations

For instance, an example of the roofline model as applied on a GPU against to 4 types of problems is depicted in figure 3.1. The operational intensity of these problems is indicative as provided by authors proposed the roofline model in their original work[92]. The chart depicts the GTX Titan-X GPU theoretical specifications. In this particular example the SpMV (Sparse Matrix-Vector multiplication) and stencil computation problems are classified as memory bound whereas the LBMHD (Lattice Boltzmann Magnetohydrodynamics) and 3D FFT problems as compute bound.

However, it should be noted that the roofline model provides an upper bound on performance by definition. There is a wide range of causes that can have a negative impact on performance on GPUs. These include bad memory access patterns, limited parallelism, inefficient kernel launch configurations, branch divergences, resource limitations (e.g. shared memory), bank conflicts (shared memory), and so on [73, 72]. In this regard, as GPUs are massively parallel compute devices, it is assumed that satisfactory parallelism is provided in order to keep the GPU busy. In general, a GPU kernel can be either memory bound, compute bound or latency bound. It is memory bound when the memory bus is congested by posing a limiting factor on the rate of execution of compute instructions. In this case the ALUs (arithmetic logic units) get stalled waiting for the memory data transfers to complete. On the other hand, it is compute bound primarily when the ALUs (Arithmetic Logic Units) of the GPU cores, or Streaming Multiprocessor in terms of CUDA, are fully utilized and unable to provide additional throughput. In case the pipeline or memory latencies are the primary reason that limit performance the kernel is considered as latency bound.

Due to the different architectural features of CPUs and GPUs some observations can be

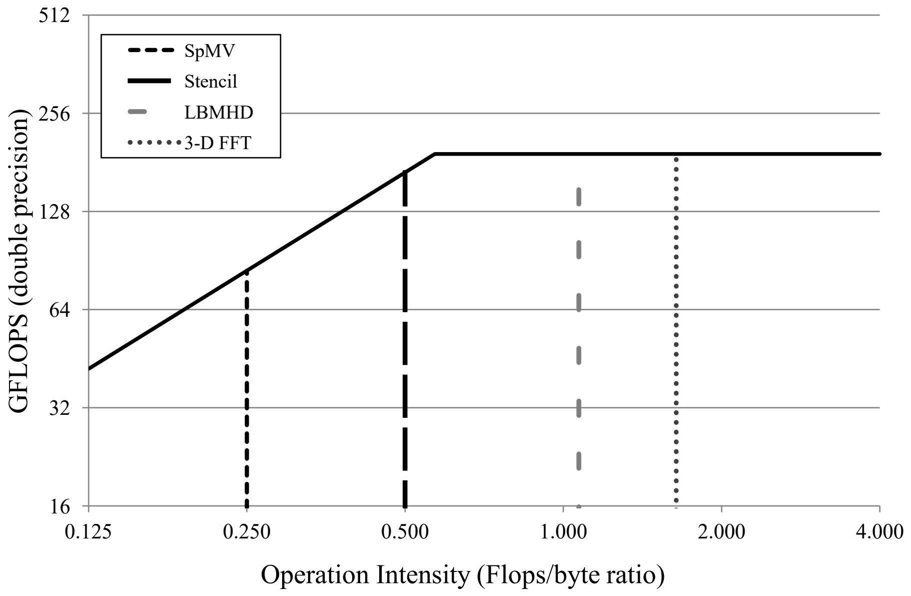


Figure 3.1: The roofline visual model on NVidia Titan X GPU against 4 applications.

made. On one hand CPUs mostly focus on reducing memory access latencies by employing a high hierarchy of large amount of memory caches. As long as the memory caching mechanism is unable to accommodate memory accesses, CPUs mostly exploit the ILP (Instruction Level Parallelism) in order to overlap computation and memory transfers. In addition, when hardware multithreading is supported, CPUs are able to overlap execution of multiple threads. Typically, however, hardware multithreading on CPUs is limited, e.g. 2 threads per CPU core on Intel CPUs (i.e. a technology marketted as *hyper-threading*). Only few server oriented CPUs provide wider multithreading capabilities (e.g. Sun UltraSPARC T1 with 4 hardware threads per core[41]) but even in these cases the amount of hardware threads is not comparable to GPUs. On the other hand, GPUs are not equipped with large cache hierarchies and the large amount of active threads mostly diminish any chance of temporal locality on memory accesses to be translated to cache hits. Therefore, they mostly rely on overlapped execution in order to hide the memory access latencies.

3.1.1 The latency hiding opportunity on GPUs

A GPU can support a vast amount of active threads counted in thousands which serves as an opportunity for hiding latencies. Active threads are handled by hardware by switching between them whenever a thread is stalled due to a memory access operation or pipeline dependency. This allows to continue execution on a thread that is available for execution in the active thread pool. Since the thread switch is done through hardware it is implemented to be as fast as possible. As long as the compute workload provides enough computations the memory access latency could potentially be hidden. It should be noted that in this context the notion of threads correspond to the *warp* on NVidia GPU platform as the instruction execution is scheduled on a per *warp* basis instead of a CUDA thread which corresponds to the data lane within a *warp*.

For instance, an NVidia GTX-980 GPU is equipped with 32 SMs, each comprised of 128 SPs[70]. Each SM employs 4 instruction schedulers, each orchestrating its own *warps*. Each scheduler gets assigned a maximum of 16 *warps* and thus, each SM is able to handle up to a total of 64 *warps* (2048 CUDA threads). This allows a maximum of 16 warps to be available in a *warp* pool for each on each scheduler, on each execution cycle, able to execute whenever the currently executing *warp* encounters a stall situation which would otherwise incur idle cycles.

Similarly, the recent AMD GPU architecture (AMD GCN[2]) allows each Compute Unit to feature 4 SIMD units, each holding its own *wavefront* pool. Each SIMD unit can keep active up to 10 total *wavefronts*. As each *wavefront* comprises of 64 *work-items* (the equivalent of a CUDA thread on AMD platform), each CU can support up to 2560 total *work-items*.

GPUs are focused on throughput instead of reduced latency of a particular thread. On ideal cases where the GPU has a large amount of parallel workload at its disposal it should be able to hide the corresponding latencies. This large amount of active threads supported by modern GPUs allows the intensive use of the roofline principle to the same or greater extent compared to a CPU. This is justified by the fact that CPUs rely mostly on single thread execution and in case of a on-chip cache miss the overlapping opportunities are limited by the amount of independent instructions in the executed instruction window. On the other hand, memory latencies for GPUs are larger and the overlapped compute-memory execution becomes highly significant.

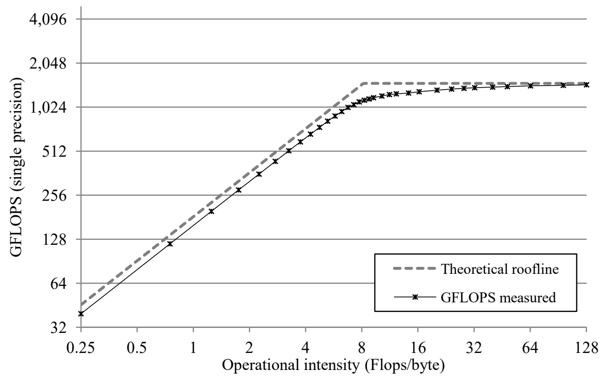
3.2 An experimental roofline approximation

In order to examine the behavior of the GPU on various operational intensities and to check the validity of the roofline model on GPUs, a micro-benchmark was developed. The micro-benchmark kernel involves both computation and memory traffic in a configurable balance inducing an artificial workload with mixed type of operations. As such, the behavior of various GPUs can be investigated in mixed types of instruction streams. In its design threads perform a small fixed number of read accesses and a configurable number of multiply-add operations. The number of compute operations is set at compilation time so the induced instruction overhead is kept to minimum. All additional instructions beyond the required ones were minimized in order to keep extra overhead as low as possible. Template variables have been used where beneficial including access strides, block size, operational intensity factor and thread coarsening factor, enabling loops to be fully unrolled [72]. The developed micro-benchmark (*mixbench-cuda-ro*) is publicly available for experimentation[1].

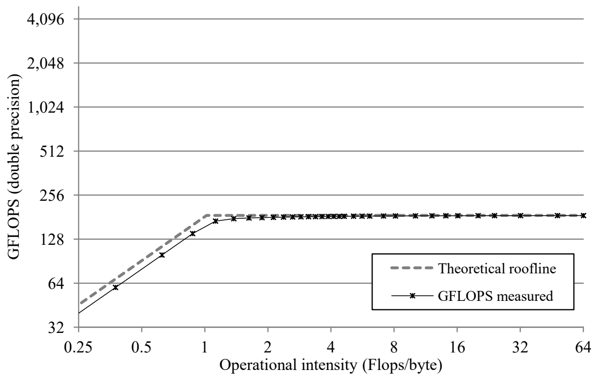
[1]`http://github.com/ekondis/mixbench/releases/tag/v0.02`

Table 3.1: Theoretical GPU specifications and the respective flops/byte ratios

	Memory	DP Compute		SP Compute	
GPU	(GB/sec)	(GFLOPS)	(Flops/byte)	(GFLOPS)	(Flops/byte)
GTX-480	177	168	0.949	1,345	7.599
GTX-660	144	83	0.576	1,983	13.771
GTX-960	112	81	0.722	2,593	23.110
GTX-1060 6GB	192	120	1.074	3,855	34.364
Tesla S2050	148	514	3.473	1,028	6.946
Tesla K20c	208	1,174	5.644	3,522	16.933



(a) SP floating point



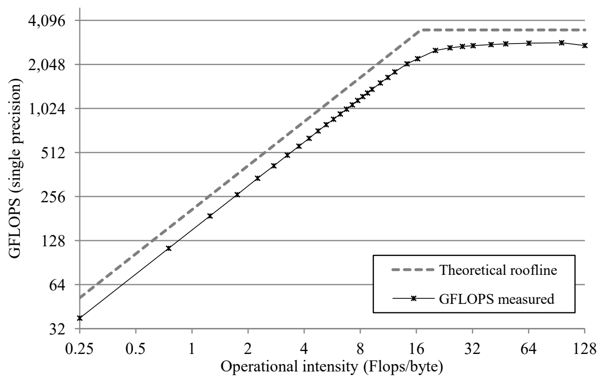
(b) DP floating point

Figure 3.2: Experimental roofline estimation on GTX-480 GPU

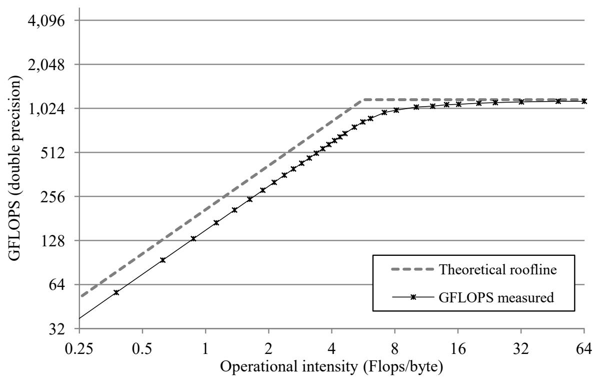
3.2.1 Experimental results

The GPU specifications for devices used in this experiment are are provided in table 3.1. Four of the GPUs are consumer parts (GeForce GTX) and the rest two are professional compute oriented parts (Tesla)[67]. All GPUs are manufactured by NVidia and support the CUDA programming environment. The GTX-480 and Tesla S2050 are based on the Fermi architecture[66], the GTX-660 and Tesla K20c are based on the Kepler architecture[69], the GTX-960 is based on Maxwell architecture[70] and the GTX-1060 is based on Pascal architecture[76].

In figure 3.2 the results of execution using the micro-benchmark on three GPUs are illustrated. The dashed line represents the theoretical peak of performance as determined by the GPU specifications. The observed performance follows a similar pattern to the theoretical one as indicated by the roofline. In particular, the GTX-480 performance approached the theoretical compute peak very closely. For the memory bound region the observed performance is a little lower than what the theoretical roofline represents and this fact expresses the inability to approach the maximum theoretical memory bandwidth on real kernels. Though the theoretical peak of the GTX-480 used in the experiments

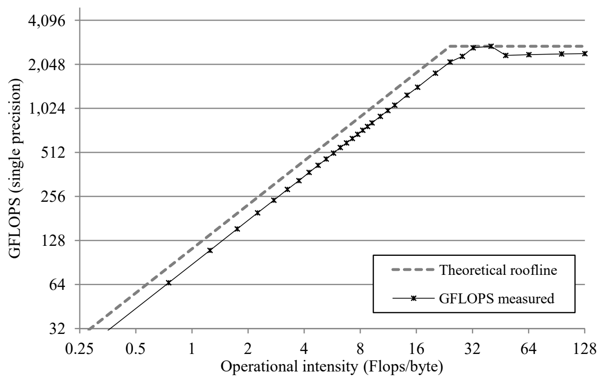


(a) SP floating point

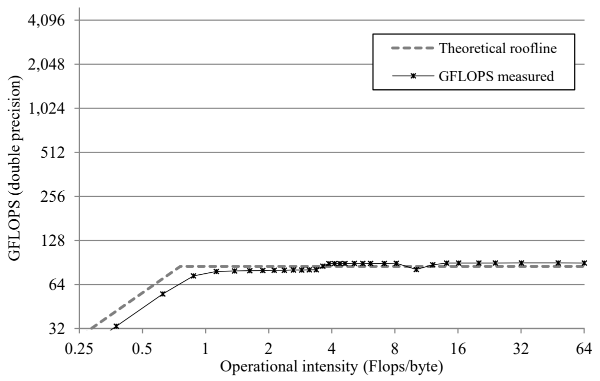


(b) DP floating point

Figure 3.3: Experimental roofline estimation on Tesla K20c GPU



(a) SP floating point



(b) DP floating point

Figure 3.4: Experimental roofline estimation on GTX-960 GPU

was 182GB/sec (the particular GPU provided overclocked memory frequencies leading to increased memory bandwidth compared to the 177 GB/sec provided by the reference GPU), the attained bandwidth did not exceed 163GB/sec. For the Tesla GPUs the performance on memory bound kernels is significantly worse due to the overhead of ECC protection[68]. In figure 3.3 the results for the Tesla K20c are depicted. It is evident that the gap between the theoretical roofline and the experimentally constructed one is larger than in the previous case. The execution results on a GTX-960 are illustrated in figure 3.4. In this case the compute performance in some cases exceeds the theoretical one as prescribed by the device specifications. This is particularly obvious in the double precision experiment and it is attributed to the dynamic clock frequencies and their ability to exceed its base clock frequency (boost clock frequency). Modern GPUs tend to work on a wide frequency scale, regularly exceeding its base clock settings, as long as its thermal/power limits are met. The GTX-960 used for the experimentation featured a 1,329MHz clock frequency as it was reported through CUDA device properties and this frequency has been used for the construction of the roofline chart. According to our measurements the particular GPU could reach up to 1,405MHz by using dynamic clock frequency adjustment. This frequency corresponds to 89.9 GFLOPS double precision theoretical peak performance and thus, justifies the observed 89 GFLOPS measurement. Similarly, the boost clock frequency of the GTX-1060 GPU is significantly higher than the base clock frequency. Table 3.1 provides the theoretical specifications based on the base clocks for the GPUs. In particular, the base clock frequency of the GTX-1060 GPU is 1,506 MHz, which corresponds to a theoretical peak of 3,855 GFLOPS (single precision). The boost clock frequency of the GPU is 1,708 MHz and this corresponds to 4,372 GFLOPS peak performance. In practice the clock frequency has been observed to even exceed the boost clock frequency in many cases. This fact poses an uncertainty even to the estimation of pure theoretical rates of modern GPUs.

In summary, the experimental approximation of the roofline model is validated. There are some observations though, such as the measured effective memory bandwidth not reaching the theoretical one. In addition, the effective compute throughput for the case of Tesla K20c GPU is significantly lower than the theoretical peak. These observations will be used in the next section for the approximation of performance of real world kernels.

3.3 The *quadrant-split* visual representation

The roofline visual model is a valuable abstract representation of the compute device capability. It can be used to map a program's operational intensity to the maximum expected theoretical performance of the device. Inherently, the roofline representation is focused on a single device on multiple problems.

As an alternative representation, the *quadrant-split* is proposed where in the horizontal axis the memory bandwidth is used instead of the operational intensity. In this respect, a device can be represented by a single point on the chart determined by its memory bandwidth and compute throughput peak rates. A program can be represented by a half-line crossing the intersection of the axes with a slope equal to its operational intensity. Typical memory bound problems tend to have a small slope whereas compute bound problems have high slope. The half-line splits the quadrant space into two half-quadrants where device points residing into the upper one have higher compute resources than memory bandwidth potential with respect to the application's requirements whereas the device points residing on the lower one have lower compute potential. Simply put, the half-line is the visual bound for the distinction of the area into two parts where the kernel is expected to behave as memory bound for the devices residing in the upper half-quadrant and as compute bound for the others instead. In this regard it is clear that a problem can be considered as memory bound for some devices and as compute bound for others. In other words the limiting factor is a relative term which is dependent on both the problem and the device specifications.

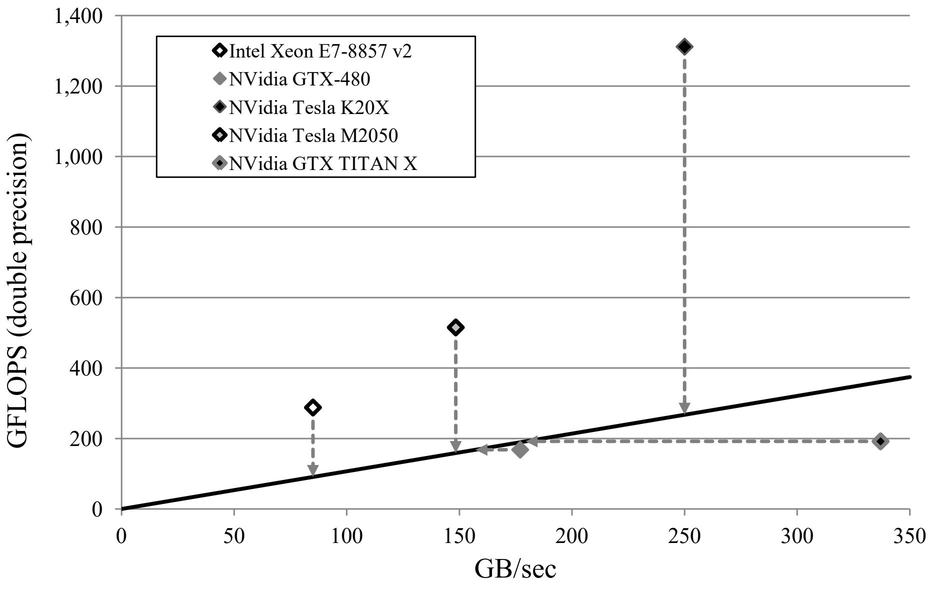


Figure 3.5: The *quadrant-split* representation of the LBMHD problem using 5 CPU/GPUs.

For instance, figure 3.5 represents the LBMHD problem with respect to 4 GPUs and a CPU. The dashed arrow lines point to the estimated roofline performance points for the each device on the particular problem. The Intel Xeon and both NVidia Tesla GPUs points reside on the upper half-quadrant which entails that the problem is memory bound with respect to these devices. In contrast, considering the GTX-480 and GTX-Titan X GPUs the problem can be considered as compute bound as their points reside on the lower half-quadrant. In order to determine the expected peak performance of a memory bound problem, a vertical line is traversed from the device point straight down to the kernel half-line (shown as a dotted line). Similarly, in order to determine the performance of a compute bound problem a horizontal line is traversed from the device point to the left till the application half-line is crossed. The intersection of these lines set the roofline performance on this device on the particular problem.

In the *quadrant-split* model more devices than one can be naturally represented on a single chart representation. Roofline model is device-centric as it is convenient for applying multiple problems on a single device whilst the *quadrant-split* model is application centric clearly depicting one application with many devices having different characteristics.

4. TOWARDS A QUANTITATIVE PERFORMANCE MODEL FOR GPUS

On the previous chapter the roofline performance of a GPU device was experimentally approximated by using a custom developed micro-benchmark application. The exhibited performance reached close to the theoretical peaks for a wide range of operational intensities in a variety of GPU devices. In this regard the theoretical roofline performance of a device was experimentally approximated with micro-benchmarking. This approximation yielded adequate results, which in many cases reached very close to the theoretically provided information. This was possible as the micro-benchmark application was designed by taking into account various aspects of the underlying architectures. First, Multiply-ADd (MAD) operations were applied, which typically are the most effective types of instructions provided by GPUs. Next, overhead was kept to minimum by eliminating factors that would increase the amount of control and address computation instructions. The workload assigned per thread was intentionally high by using thread coarsening techniques, plus template variables and loop unrolling methods were intensively applied. Memory access patterns were also designed to be effectively coalesced. Last, the amount total threads was also high in order to ensure that all SMs were kept highly utilized. All these factors pushed the micro-benchmark's performance to approach very close to peak theoretical performance.

Since this was proved possible the question arising is to which extent the opposite procedure is feasible, i.e. estimating the performance of a particular kernel by using the theoretical specifications or other metrics of a device. This is the original value of the roofline model, which is based mostly on theoretical specifications and thus, provides qualitative characterization of programs. Knowing the amount of compute operations conducted by a kernel and the respective memory traffic should give a sense of the expected program behavior but there are also other essential factors affecting performance. What is the performance of the GPU on the particular kernel's compute operations? Does the memory subsystem perform as expected? What is the effect of the rest instructions executed on the execution time? In order to provide useful predictions such questions raise issues that the design of a practical performance model should take into account.

To this direction in this chapter, after investigating the most significant factors leading to performance degradations, a performance model is proposed focusing on the performance prediction of real world kernels. The proposed model is a throughput based roofline model which involves a set of adjustments based on both the kernel itself and the underlying device, allowing the realistic predictions of real world kernels.

4.1 Motivation and performance considerations

As already stated, the aforementioned micro-benchmark is tailored to yield the highest raw performance in compute throughput as measured in FLOPS (Floating Point Operations) or memory transfer performance when it comes to memory bandwidth. This was achieved by focusing on a micro-benchmark design which provides the necessary freedom to apply strict optimizations. While this is possible on a micro-benchmark, on real world kernels applying similar optimizations to the same extent is often is not possible.

For instance, the instruction overhead typically consists a much larger portion of the executed instruction stream and thus, the induced overhead is significant. The cost of instruction execution comprises by both compute operations and other overhead instructions. The overhead instructions can be address calculations, control instructions and possibly load/store instructions. Control instructions include integer calculations, comparisons, branches (conditional or not), intra-block synchronization, etc. Load/store instructions are used to access all possible memory spaces including global, shared or texture memory. They are handled by separate execution units which typically count less than the Streaming Multiprocessors per Multi-Processor. The cost of executing these additional types of instructions should be considered by a performance model in order to be more accurate.

In addition, memory coalescing is not always practical on irregular applications or not always adequately applied by the programmer. The majority of modern scientific applications tend to be memory bound, especially in cases where the memory accesses have not been carefully refined. Thus, the negative performance effects of imperfect coalescing should absolutely be considered in a performance model. The proposed performance estimation should therefore address these issues.

4.2 A quantitative roofline GPU performance model

Since GPUs are throughput oriented processors, in this work a throughput based approach is followed regarding performance modeling. Having the roofline model[92] as a foundation, the proposed performance model introduces and takes into account the most significant factors affecting the effective GPU compute throughput. The role of roofline model is to distinguish between the two primary involved performance limiting factors in the program's execution i.e. the compute throughput and the memory bandwidth. The impact of memory traffic is important for GPUs as the latter require being fed with vast amounts of data in order to keep their compute resources busy. This becomes more critical due to the absence of large cache hierarchies.

This work provides not only a qualitative analysis on GPU performance but quantitative results, as well. Typically, the original roofline model relies on the theoretical peak specifications of the processor. In the proposed model measurements through micro-benchmarks are employed for both compute throughput and memory bandwidth peak rates. In addition, the peak compute throughput is further adjusted to an estimated maximum rate of execution allowed by the kernel under inspection. This adjustment involves the construction of a set of kernel parameters in a "*black box*" fashion, by collecting hardware metrics using the CUDA provided profiling tools. The combination of these parameters with the GPU parameters extracted as micro-benchmark data are used to predict the expected performance of the particular kernel on target GPU. As a result, this method achieves more accurate results compared to using the pure theoretical peak values.

The primary benefits of the proposed performance model approach are its simplicity and ability to run as part of an automated tool. It is able to run in an automated fashion as it does not require user's intervention since all GPU parameters rely on measurements captured using micro-benchmarks and kernel parameters are collected using hardware profiling executions. Thus, no source code or binary code is analyzed as all data is gathered through a hardware metric profiling procedure.

4.2.1 An overview of the proposed model

The proposed performance prediction model follows the roofline model approach. In this respect, it is based on throughput of instruction execution and memory bandwidth. However, it involves a set of corrections/adjustments in order to produce more accurate quantitative results:

1. Peak performance measurements
   Although, the theoretical specifications set a good base of the performance that can be achieved on a device, this is not always feasible in practice. In some cases the measured performance is a fraction of the theoretical rate, especially in case of memory bandwidth measurements. In order to estimate the practical peaks in both compute and memory transfer performance a set of micro-benchmark kernels were developed through which the real performance of the devices under investigation is evaluated.

2. Floating point operation efficiency
   GPU vendors tend to provide the peak performance achieved using multiply-add operations. These operations fuse a multiplication and an addition operation into a single instruction ($a \times b + c$). These instructions are typically optimized to be executed in just one shader cycle (single precision throughput). The theoretical peak rates provided by vendors assume a perfectly balanced stream of floating point multiplications and additions. If the stream of executed instructions is not perfectly balanced then the performance drops. For instance, having a pure stream of addition instructions would reduce the floating point performance to a half as the addition instructions perform just one operation instead of two and they are executed as fast as the multiply-add instructions.

3. Instruction mix efficiency
   Another factor that further lowers peak floating point performance of a kernel is the instruction overhead in the executed instruction stream. As far as we focus on scientific problems the beneficial instructions are the floating point instructions which perform the actual computations as required by the algorithm. The rest of the instructions can be control flow, address calculations, operations on auxiliary integer variables (e.g. accumulators), etc. All these operations consume valuable resources of the GPU, both of the instruction scheduler and the ALUs, thus, they limit the peak floating point performance to lower levels than the theoretical ones.

The proposed model incorporates the cost induced by the execution of all types of instructions, beyond the beneficial compute instructions. In this respect, a profiling approach on a reference GPU is employed by extracting kernel execution information without requiring any internal knowledge of the kernel characteristics. This method allows the actual memory traffic generated by the kernel to be implicitly considered in the performance model. This includes all potentially uncoalesced memory operations that cause additional memory traffic to memory subsystem. The parameters used for the GPU device that is targeted for performance prediction are extracted by running a set of micro-benchmarks. An additional benefit of this approach is the ability to run the whole prediction tool-chain without any further input or intervention from the user. The whole process involves the steps described in figure 4.1.

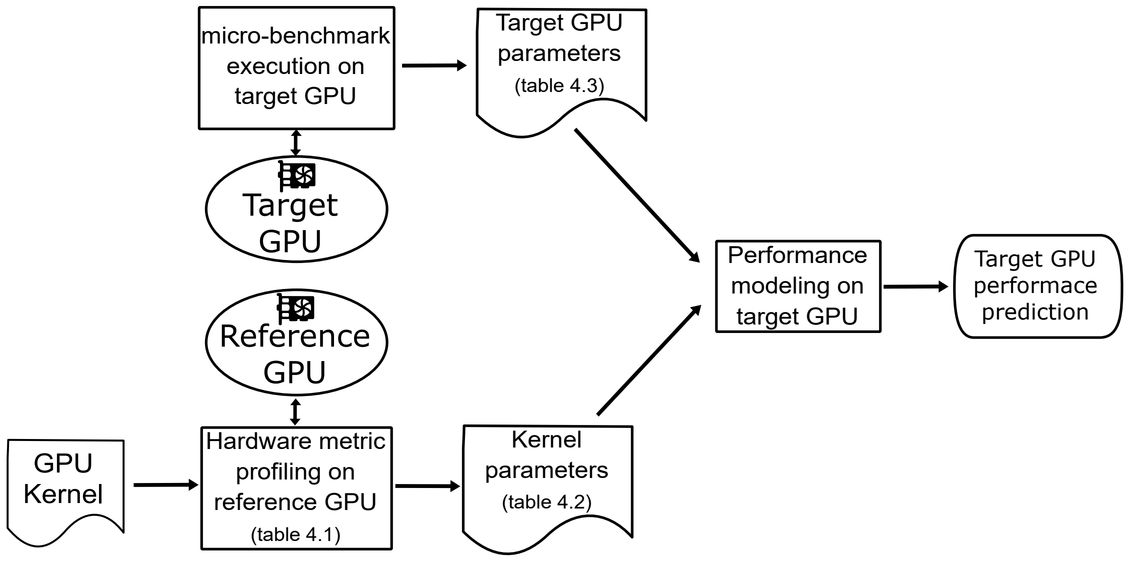

micro-benchmark execution on target GPU
Target GPU parameters (table 4.3)
Target GPU
Reference GPU
Performance modeling on target GPU
Target GPU performace prediction
GPU Kernel
Hardware metric profiling on reference GPU (table 4.1)
Kernel parameters (table 4.2)

Figure 4.1: The performance prediction methodology flow diagram.

In general, the approach for performance estimation of GPU kernels can be summarized in three aspects:

- Modeling compute and memory parameters of GPU kernels, largely independently of GPU architectural details, obtained by using a “*black box*” approach based exclusively on profiling measures (figure 4.1: ”Hardware metric profiling on reference GPU”)
- Modeling the GPU generic peak performance ratings on various operations, obtained by micro-benchmarking the target GPU (figure 4.1: ”Micro-benchmark execution on target GPU”)
- Estimation of the target GPU performance (figure 4.1: ”Performance modeling on target GPU”) on the particular kernel according to:
  - the estimated maximum rate of executed compute operations on the target GPU for the particular kernel and

– the compute and memory demands of the given kernel (i.e. operational intensity) determining whether its performance is limited by the compute or memory throughput when executed on the target GPU

4.2.2 Kernel parameter extraction

The required kernel parameters are extracted by profiling the execution of the subject kernel on a reference GPU. The *nvprof* NVidia profiling utility[71] is used which provides a rich set of available GPU metrics in order to shed light on the kernel's execution process. The list of the required kernel metrics is shown in table 4.1, with the description provided by the *nvprof* utility documentation[71]. For the rest of this thesis the notation of the metric will be used for reference. The $M_{tran\text{-}r}$ and $M_{tran\text{-}w}$ metrics reflect the induced DRAM memory traffic (including the overhead memory traffic induced by sub-optimal coalescing) and all the rest of metrics reveal information regarding the instructions executed by the kernel.

Table 4.1: The NVidia GPU profiler metrics required for the derivation of kernel parameters.

Metric	Notation	Description
flop_count_sp_fma	M_{fma32}	Number of single-precision floating-point multiply-accumulate operations executed by non-predicated threads
flop_count_dp_fma	M_{fma64}	Number of double-precision floating-point multiply-accumulate operations executed by non-predicated threads
inst_compute_ld_st	M_{ldst}	Number of compute load/store instructions executed by non-predicated threads
inst_executed	M_{inst}	The number of instructions executed
inst_fp_32	M_{fp32}	Number of single-precision floating-point instructions executed by non-predicated threads (arithmetic, compare, etc.)
inst_fp_64	M_{fp64}	Number of double-precision floating-point instructions executed by non-predicated threads (arithmetic, compare, etc.)
inst_integer	M_{int}	Number of integer instructions executed by non-predicated threads
dram_read_transactions	$M_{tran\text{-}r}$	Device memory read transactions
dram_write_transactions	$M_{tran\text{-}w}$	Device memory write transactions

The produced parameter set is provided in table 4.2. K_{type} parameter determines the type of beneficial operations within the kernel. It can be either *fp64*, *fp32* or *int*. A simple rule based approach in order to avoid user interaction is a function selecting *fp64* if the M_{fp64} metric is non zero, *fp32* if the M_{fp32} is non zero or *int* otherwise. The type of instructions determined by K_{type} is considered as the one that clearly contributes to the actual computation and all the rest instructions are considered as overhead. The W_{comp} parameter

Table 4.2: The set of required *kernel parameters* in the proposed performance model.

Parameter	Description	Obtained
K_{type}	Dominant ops (fp64, fp32 or int)	rule based function
W_{comp}	Compute operations	formula (4.1)
W_{traf}	DRAM bytes accessed	formula (4.2)
E_{mix}	Operation mix efficiency (%)	formula (4.3)
D_{ops}	Operation instruction density (%)	formula (4.6)
D_{ldst}	Ld/St instruction density (%)	formula (4.7)
D_{other}	Other instruction density (%)	formula (4.8)

represents the total beneficial compute operations performed by the kernel. It's equal to the amount of kernel operations that are specified to be of type K_{type}. It is evaluated by using formula (4.1).

$$W_{comp} = \begin{cases} M_{fp32} + M_{fma32}, & \text{if } K_{type} = \text{fp32} \\ M_{fp64} + M_{fma64}, & \text{if } K_{type} = \text{fp64} \\ M_{int}, & \text{if } K_{type} = \text{int} \end{cases} \tag{4.1}$$

It should be noted that there are also metrics provided for the floating point operations (*flop_count_sp* and *flop_count_dp* metrics) but some floating point instructions are excluded by this metric and thus, it was chosen to use the indirect formula (4.1) for the estimation of compute operations. In any case, the expected divergence on the results is minimal.

The parameter regarding the conducted memory traffic is the W_{traf}. It is estimated by using the DRAM transaction count metrics and shown in formula (4.2) as the size of each transaction on NVidia platform is currently 32 bytes.

$$W_{traf} = 32 \times (M_{tran\text{-}r} + M_{tran\text{-}w}) \tag{4.2}$$

As it has been already noted, not all compute instructions perform the same amount of operations. The instruction that is typically used in GPU performance specifications is the Multiply-Add instruction which performs 2 operations per instruction, one multiplication plus one addition. Other instructions perform a single operation in general. Therefore, the efficiency of compute instructions E_{mix} is defined as 100% when all compute instructions perform 2 operations each or 50% when all compute instructions perform just a single operation. In real world kernels it ranges from 50% to 100% depending on the usage of multiply-add operations. In this respect, it is estimated by using formula (4.3) which involves the type of compute instructions executed.

$$E_{mix} = \begin{cases} \frac{M_{fp32}+M_{fma32}}{2\times M_{fp32}} \times 100\%, & \text{if } K_{type} = \text{fp32} \\ \frac{M_{fp64}+M_{fma64}}{2\times M_{fp64}} \times 100\%, & \text{if } K_{type} = \text{fp64} \\ 50\%, & \text{if } K_{type} = \text{int} \end{cases} \tag{4.3}$$

Finally, the instructions executed are classified in 3 different types (compute, load/store and other instructions) and the individual density of each type in the instruction stream is determined. In this regard, the compute instruction count is divided by the total instruction count as formulae (4.4), (4.5) and (4.6) indicate. In the same manner, the load/store instruction count is used in the calculation of load/store instruction density, shown in formula (4.7). The percentage of other instructions is the complement of the first two densities, as shown in formula (4.8).

$$I_{ops} = \begin{cases} M_{fp32}, & \text{if } K_{type} = \text{fp32} \\ M_{fp64}, & \text{if } K_{type} = \text{fp64} \\ M_{int}, & \text{if } K_{type} = \text{int} \end{cases} \tag{4.4}$$

$$I_{total} = 32 \times M_{inst} \tag{4.5}$$

$$D_{ops} = \frac{I_{ops}}{I_{total}} \times 100\% \tag{4.6}$$

$$D_{ldst} = \frac{M_{ldst}}{I_{total}} \times 100\% \tag{4.7}$$

$$D_{other} = 100\% - D_{ops} - D_{ldst} \tag{4.8}$$

4.2.3 Target GPU parameter extraction

All required device parameters are collected by using micro-benchmarks. The list of parameters is shown in table 4.3. All floating point computation throughput parameters (T_{SP} and T_{DP}) concern MAD (Multiply-ADd) operations. The T_{xxx} parameters (T_{SP}, T_{DP}, T_{int}, T_{add}, T_{ldst}) regard the compute throughput of the device in various types of instructions and the B_{mem} parameter which reflects the effective memory bandwidth of the device.

The T_{SP}, T_{DP}, T_{int}, T_{add} metrics are collected by using a micro-benchmark that excessively performs execution of independent arithmetic instructions. The T_{ldst} metric is measured by using a micro-benchmark that performs intense memory copies between memory locations within shared memory, which has been presented along with 2 additional micro-benchmarks in [43]. Typically, shared memory which is a scratchpad, exhibits the highest

Table 4.3: The set of *GPU parameters* used in the performance model.

Parameter	Description	Unit
T_{SP}	Single precision floating point operation throughput	GFLOPS
T_{DP}	Double precision floating point operation throughput	GFLOPS
T_{int}	Integer multiply-add operation throughput	GIOPS
T_{add}	Integer addition operation throughput	GIOPS
T_{ldst}	Load/Store instruction throughput on shared memory	GOPS
B_{mem}	Memory bandwidth	GB/sec

performing memory type accessed by load/store instructions. More information regarding this micro-benchmark is provided in B.2 section of the appendix. Last, the B_{mem} metric is estimated by applying memory operations like reads, writes and copies on large arrays. The average bandwidth on these three operations is selected for each device.

As an alternative, in case that the target GPU hardware is not accessible (or even non existing) these parameters could be determined by using rough estimations through the specifications and the GPU's architecture documentation. It is common for vendors to provide throughput characteristics in operations/clock per SM[73].

4.2.4 Kernel performance estimation

At this point, the efficiency of instruction execution regarding beneficial computation is modeled. In this model the throughput of various instruction types is considered. As the instruction pipeline is occupied for the execution of various types of instructions, the pipeline is only partially available for the execution of beneficial instructions. Thus, the purpose is to estimate the attainable peak throughput by considering the portion in which the pipeline is available for the execution of beneficial instructions.

GPUs exhibit varying execution throughput depending on the type of instruction under execution. For instance, the Tesla K20c can execute a third of the floating point operations in double precision compared to SP in the same amount of time. In this regard the instruction type densities (D_{ops}, D_{ldst}, D_{other}) should be considered in order to provide an estimation on the overall instruction execution throughput on the particular kernel.

The peak throughput on raw beneficial operations is selected in (4.9) according to K_{type}:

$$T_{op} = \begin{cases} T_{SP}, & \text{if } K_{type} = \text{fp32} \\ T_{DP}, & \text{if } K_{type} = \text{fp64} \\ T_{int}, & \text{if } K_{type} = \text{int} \end{cases} \tag{4.9}$$

For the estimation of the instruction execution efficiency the instruction densities along with the instruction throughput for various types are considered. The instruction types considered correspond to the throughput parameters of the GPU (table 4.3). The fastest instruction on the GPU typically is the single precision multiply-add instruction, and therefore it is the instruction that potentially is used to execute the most operations per second. So, the single precision multiply-add instructions are used as a point reference. The weight factor of executing a type of instruction is defined as the throughput ratio of fast single precision floating point instructions to the throughput of the particular type of instructions. Thus, weight factor is normalized by setting the weight of single precision instructions to 1. Therefore, the weight of all other instructions is typically greater or equal to 1. A simplified interpretation of the weight factor is that it represents a measure proportional to the time that the instruction execution keeps the pipeline busy relative to the time that a fast single precision instruction does. In this regard we define the weight factor operators as follows in formulae (4.10), (4.11), (4.12):

$$W_{op} = \frac{T_{SP}}{T_{op}} \tag{4.10}$$

$$W_{ldst} = \frac{{}^{1}\!/_{2} T_{SP}}{T_{ldst}} \tag{4.11}$$

$$W_{other} = \frac{{}^{1}\!/_{2} T_{SP}}{T_{add}} \tag{4.12}$$

In the estimation of W_{other} the throughput of integer addition is used. This is an arbitrary decision based on the assumption that the rest of the instructions apart from computation and load/store, is constituted mostly of simple integer instructions or instructions that execute roughly with the same cost. The ½ factor in (4.11) and (4.12) is applied in order to convert the operation throughput rate T_{SP} to instruction execution rate as each floating point MAD instruction is accounted as 2 operations. All beneficial operations are assumed to be executed using MAD instructions (two operations per instruction) whereas the load/store and integer addition operations are assumed to be implemented with single operation instructions. By taking into account the instruction densities and the respective weight factors the relative execution cost of each instruction type can be defined as shown in (4.13), (4.14), (4.15):

$$C_{op} = D_{ops} \times W_{op} \tag{4.13}$$

$$C_{ldst} = D_{ldst} \times W_{ldst} \tag{4.14}$$

$$C_{other} = D_{other} \times W_{other} \tag{4.15}$$

The load/store instruction cost should not be confused with the load/store DRAM throughput. It models the instruction throughput of the load/store unit, which mostly depends on the amount of load/store units per multiprocessor. Though not always all instructions in a category exhibit the same throughput (e.g. integer addition vs integer shift operation) there are no hardware metrics providing more detailed classification of instructions [71]. Thus, this is a compromise that is forced by both the available hardware metrics and as a decision in order to restrict the model complexity. The estimated instruction efficiency can be estimated as given by formula (4.16):

$$E_{instr} = \frac{C_{op}}{C_{op} + C_{ldst} + C_{other}} \times 100\% \tag{4.16}$$

This cost modeling for the instruction execution assumes that all instructions are executed by the GPU multiprocessor on a single pipeline and therefore the execution of different types of instructions cannot be co-issued in a super-scalar fashion. Although this assumption is not accurate, we argue that the instructions in general are not co-issued. For instance, multiprocessors based on Kepler architecture can issue 8 instructions per cycle (4 instructions per scheduler, each SM having 2 schedulers) while only 6 instructions can be issued to the ALU pipes ($32 \times 6 = 192$ SPs). In this case, two additional instructions could potentially be issued to the Load/Store units on the same cycle. However, this ideal case is not typical and therefore in this model it is assumed that the SM consists of a single instruction pipeline. An investigation of the multi-pipeline model is left as future work.

The adjusted throughput is estimated by applying both the efficiency ratios each decreasing the theoretical instruction throughput by a factor. The adjusted throughput is given in (4.17):

$$T'_{op} = E_{mix} \times E_{instr} \times T_{op} \tag{4.17}$$

As such, the kernel's operational intensity is $O_{krn} = W_{comp}/W_{traf}$ and the device's adjusted operational intensity is $O_{dev} = T'_{op}/B_{mem}$. The comparison of the two values is used to determine if the application is considered to behave as memory bound or compute bound. If $O_{krn} > O_{dev}$ then the kernel is considered as compute bound for the particular device or memory bound otherwise. Thus, the estimated compute throughput is given by (4.18):

$$T_{predicted} = \begin{cases} T'_{op}, & \text{if } O_{krn} > O_{dev} \\ O_{krn} \times B_{mem}, & \text{if } O_{krn} \leq O_{dev} \end{cases} \tag{4.18}$$

4.3 Case study 1: Red/black SOR stencil computation

In order to make the proposed method easier to understand, in this section a case study is presented along with the execution of all required steps as prescribed in the previous section. The problem on which the performance prediction method is applied is a stencil computation. More specifically, the SOR method is applied with red/black ordering, which effectively performs iterative computations between neighboring elements on a 8192x8192 2D grid (figure 4.2) comprised of double precision values. The method is described in detail in the next chapter (5.1.1) accompanied with a wide range of experimental results. Furthermore, related work has also been published by the author of this thesis [44, 45] and can be referred for more information.

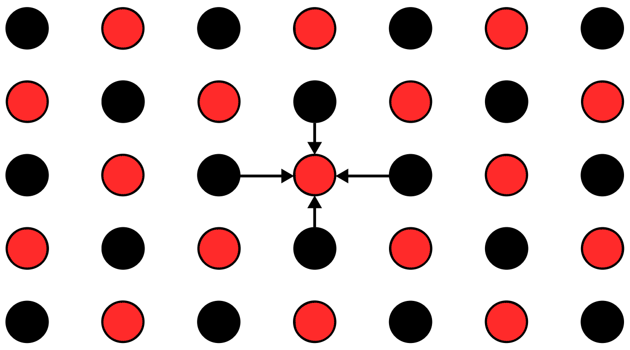


Figure 4.2: Red points depend only on the neighboring black points and vice versa.

After running 4 iterations of the computation on the GTX-480, the accumulated metrics were captured with the *nvprof* tool for all iterations of the red elements' computation as shown in table 4.4. Thereafter, the kernel parameters can be determined by the equations (4.1-4.8) as depicted in table 4.5. As such, the kernel's operational intensity is $O_{krn} = \frac{W_{comp}}{W_{traf}} \approx 0.3$ which is considered to be low. In addition, the instruction overhead as expressed by D_{other}, occupies the majority of the instruction stream (figure 4.3). However, if the expected nature of the problem being memory bound is true then the overhead is expected to be hidden.

Table 4.4: The profiling metrics gathered for the red/black computation on a GTX-480 GPU.

Parameter	Measurement
M_{fma32}	0
M_{fma64}	33,554,432
M_{ldst}	303,079,424
M_{inst}	56,100,732
M_{fp32}	0
M_{fp64}	218,107,904
M_{int}	736,891,392
$M_{tran\text{-}r}$	17,660,604
$M_{tran\text{-}w}$	8,392,704

Parameter	Value
K_{type}	fp64
W_{comp}	1,006,649,344
W_{traf}	3,334,823,424
E_{mix}	57.69%
D_{ops}	12.15%
D_{ldst}	16.88%
D_{other}	70.97%

Table 4.5: The extracted kernel parameters of the red/black SOR kernel

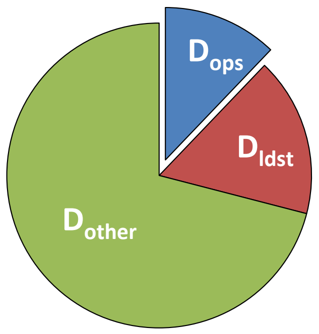


Figure 4.3: Instruction densities per instruction type for the red/black SOR kernel

After having collected the kernel's parameters, the device throughput parameters have to be estimated in order to run the model. After running the micro-benchmarks, the GPU parameters as shown in table 4.6 were assessed for the GTX-660 GPU. These parameters will be used for the performance prediction of the case studies in this chapter.

Table 4.6: Measured *GPU parameters* for the NVidia GTX-660

Parameter	Value	Unit
T_{SP}	1,940.80	GFLOPS
T_{DP}	89.70	GFLOPS
T_{int}	359.04	GIOPS
T_{add}	621.36	GIOPS
B_{mem}	117.56	GB/sec
T_{ldst}	169.58	GOPS

The instruction weight factors are estimated by using formulae (4.9)-(4.12) along with the GPU parameters (table 4.6). These are shown in table 4.7. These weights represent a comparison of throughput in each operation with respect to the throughput in single precision operation execution (T_{ops}/T_{SP}). For instance, the consumer Kepler GPUs feature a quite low double precision operation throughput rated a 1⁄24 of their single precision operation throughput. This is based on the fact that each SM on consumer Kepler GPUs is equipped with just 8 double precision SPs compared to 192 single precision SPs. This is approximated by weight $W_{op} = 21.64$ which means that the measured throughput of single precision operations was 21.64 times higher than the measured throughput in double precision operations. As such, weight factors represent the internal balance of available resources in the GPU multiprocessor per type of operation.

The rest of this section will be focused on the performance estimation on the GTX-660. The relative execution cost per instruction type can be estimated using equations (4.13), (4.14) and (4.15) with the weight factors provided for the GTX-660. These are $C_{op} \approx 2.63$, $C_{ldst} \approx 0.97$ and $C_{other} \approx 1.11$ respectively. Using formula (4.16) we get $E_{instr} = 55.89\%$. This rate expresses an approximation of the instruction execution cost of beneficial compute instructions (double precision in this case) in the whole instruction stream. After ap-

Table 4.7: The GPU cost weights as measured in the model for the NVidia GTX-660

Weight	Value
W_{op} (fp32)	1.00
W_{op} (fp64)	21.64
W_{op} (int)	5.41
W_{ldst} (load/store)	5.72
W_{other} (add)	1.56

plying the efficiency factors (E_{mix} and E_{instr}) to get the adjusted throughput using equation (4.17) we get $T'_{op} = 28.92$ GFLOPS. This suggests the maximum expected performance of this kernel, ignoring the required DRAM bandwidth. In order to take memory bandwidth into account the operational intensity of the device $O_{dev} = T'_{op}/B_{mem} \approx 0.25$ should be compared with the kernel's operational intensity $O_{krn} \approx 0.3$. Since the latter is higher than the former, it means that the kernel itself requires more compute throughput than the GPU can provide, given the characteristics of both the kernel and the GPU. Therefore, the kernel is designated as compute bound on the particular GPU, in contrast to the initial intuition, with $T_{predicted} = T'_{op} = 28.92$ GFLOPS.

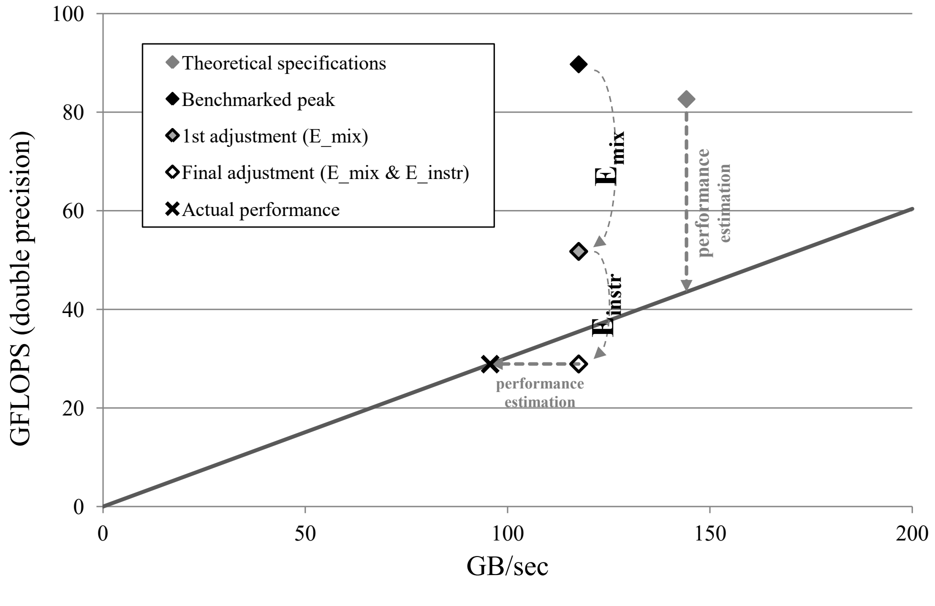


Figure 4.4: Visualization of red/black SOR performance estimation on GTX-660 with efficiency adjustments.

In figure 4.4 a visual representation is provided for applying the proposed model versus using a straightforward approach based on the GPU's theoretical specifications. The solid dark gray half-line represents the red/black SOR kernel's operational intensity (slope ~0.3 flops/byte). The gray point is determined by the theoretical specifications of the GPU. Thus, by using just the gray point to determine performance one can infer that this kernel is memory bound for this GPU, as the point resides well above the line representing the kernel's operational intensity. The roofline performance is estimated by following the vertical dashed line downwards till the point it meets the kernel's half-line and this happens at a point exceeding 43 GFLOPS. However, following the proposed approach, the starting point is set at the coordinates designated by the measured peak with micro-benchmarks (top black point). Afterwards, the performance point is adjusted twice (E_{mix}, E_{instr}) by applying the efficiency degradations and the attained performance drops to a point well below the kernel's half-line. This means that the kernel is eventually considered as compute bound using the proposed model on the particular GPU. The estimated performance does not exceed 29 GFLOPS and one can estimate memory bandwidth by following the horizontal dashed line to the left till it crosses the kernel's line. This is a particular example where the estimated performance is extremely close to the actual measured performance of the kernel as the latter is designated by the ”X” on the kernel's half-line (28.88 GFLOPS).

In order to further validate the compute/memory characterization, a study of the utilization metrics was conducted. All hardware metrics provided by the nvprof tool named with the *”_utilization”* suffix represent a rough measurement of the utilization ratio of a particular GPU resource, ranging from ”Idle(0)” to ”Max(10)”. The only exception is the *issue_slot_utilization* which provides a percentage of the issue slots that issued one or more instructions. As such we collected the utilization rates for the GPU in table 4.8. The highest utilizations potentially expose the resources that are mostly utilized. In this case the ALU function unit utilization is *High (9)* which means that the ALU units already work near their full potential. The DRAM utilization is also high but not at the same rate (*High (7)*). Therefore, the utilization metric rates also lean towards the compute bound characterization being consistent with the outcome of the proposed performance model.

Table 4.8: Utilization metric values of the red/black SOR kernel, on the GTX-660.

*Metric	Utilization description	Value
alu_fu	Arithmetic Function Unit	High (9)
dram	Device Memory	High (7)
l1_shared	L1/Shared Memory	Mid (4)
ldst_fu	Load/Store Function Unit	Mid (4)
l2	L2 Cache	Low (3)
cf_fu	Control-Flow Function Unit	Low (1)
sysmem	System Memory	Low (1)
tex_fu	Texture Function Unit	Idle (0)
tex	Texture Cache	Idle (0)

* The “_utilization” suffix from metric names has been omitted

4.4 Case study 2: SGEMM computation

As a complementary example, the method is additionally applied on a traditionally compute intensive kernel. The selected application is the SGEMM (Single precision GEneric Matrix

Multiplication). The computation is defined as in (4.19):

$$C \leftarrow \alpha A \times B + \beta C \tag{4.19}$$

However, in this kernel the core of the computation was implemented (4.20) for simplicity reasons. This is the very non-trivial part of the computation as it requires intense amount of operations and data sharing in order to achieve adequate performance. The implementation is straightforward and fundamental optimizations have been applied, i.e. memory coalescing, block tiling and data sharing using shared memory. The source code of the kernel is provided for further details in the appendix.

$$C \leftarrow A \times B \tag{4.20}$$

The matrix sizes used are 1280×640 for matrix A and 640×640 for matrix B. Kernel parameters were collected using an NVidia GTX-480 and the performance estimation provided for the GTX-660 NVidia GPU. These GPUs feature a different architecture as the first one is based on Fermi architecture and the latter on Kepler architecture.

After running the computation on the GTX-480, the accumulated metrics were captured with the *nvprof* tool by running the computation and they are shown in table 4.9.

Table 4.9: The profiling metrics gathered for the SGEMM computation on a GTX-480 GPU.

Parameter	Measurement
M_{fma32}	524,288,000
M_{fma64}	0
M_{ldst}	721,715,200
M_{inst}	46,208,000
M_{fp32}	524,288,000
M_{fp64}	0
M_{int}	164,659,200
$M_{tran\text{-}r}$	1,218,190
$M_{tran\text{-}w}$	102,400

Thereafter, the kernel parameters can be determined by the equations (4.1-4.8) as depicted in table 4.10. As such, the kernel operational intensity is $O_{krn} = {}^{W_{comp}}/_{W_{traf}} \approx 24.81$ which is considered sufficiently high. In addition, though the instruction overhead is low ($D_{other} = 15.73$%), the load store instruction portion occupies almost half of total instructions as expressed by $D_{ldst} = 48.81$% (figure 4.5). Thus, the intense use of load/store instructions poses the consideration of the latter in the model a necessity in order for the performance prediction to work adequately.

Parameter	Value
K_{type}	fp32
W_{comp}	1,048,576,000
W_{traf}	42,258,880
E_{mix}	100%
D_{ops}	35.46%
D_{ldst}	48.81%
D_{other}	15.73%

Table 4.10: The extracted kernel parameters of the SGEMM kernel

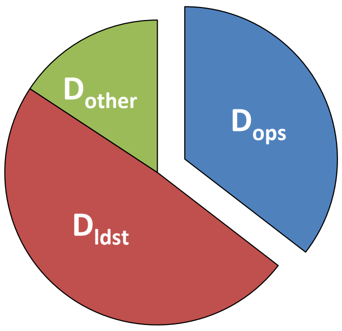


Figure 4.5: Instruction densities per instruction type for the SGEMM kernel

For this particular kernel K_{type} = fp32, thus $W_{op} = 1.00$. The relative execution cost per instruction type can be estimated by using equations (4.13), (4.14) and (4.15) with weight factors provided in table 4.7 for the GTX-660. These are $C_{op} \approx 0.35$, $C_{ldst} \approx 2.79$ and $C_{other} \approx 0.25$, respectively. Therefore, from formula (4.16) we estimate $E_{instr} = 10.45$%. This rate expresses an approximation of the instruction execution cost of beneficial compute instructions (double precision in this case) in the whole instruction stream. The more it approaches 100% the more efficient the instruction execution stream is considered. In this case this ratio is particularly low due to the high ratio of load/store instructions and the high relative cost of these instructions compared to single precision multiply-add instructions ($C_{ldst} \approx 8 \times C_{op}$).

On the other hand $E_{mix} = 100$%, which is optimum. Almost all beneficial compute instructions are multiply-add operations which was expected due to the nature of core computation of matrix multiplication as it consists of multiplications of scalars followed by an accumulation. These operations are optimally implemented on GPUs as the 2 operations (addition + multiplication) can be executed by the SP in just one cycle.

After applying the efficiency factors (E_{mix} and E_{instr}) to get the adjusted throughput using equation (4.17) it is estimated $T'_{op} = 202.80$ GFLOPS. This suggests the maximum expected performance of this kernel, ignoring the DRAM bandwidth requirements. In order to take memory bandwidth into account the operational intensity of the device $O_{dev} = T'_{op}/B_{mem} \approx 1.73$ should be compared with the kernel's operational intensity $O_{krn} \approx 24.81$. Since the latter is significantly higher than the former, it means that the kernel itself requires more compute throughput than the GPU can provide, given the characteristics of both the kernel and the GPU. Therefore, the kernel is designated as compute bound on the particular GPU, which was expected for the matrix-multiplication problem.

Similarly to the previous section, in figure 4.6 a visual representation is provided for applying the proposed model versus using just the GPU's theoretical specifications. The solid dark gray half-line represents the kernel's operational intensity (slope ~24.81 flops/byte). The gray point is determined by the theoretical specifications of the GTX-660 GPU. It is evident that performance is characterized as compute limited as the device point resides below the kernel's half-line. That said, the roofline performance is estimated by following the horizontal dashed line to the left till it meets the kernel's half-line and this happens at the point where the bandwidth falls down to ≈ 80 GB/sec. In a pure workload of $W_{ops} \approx 1.049 \cdot 10^9$ floating point operations the kernel would execute in just 0.52 msecs. However, this estimation assumes that the kernel is able to sustain the maximum theoretical performance of the device in GFLOPS which experience has shown that it is not feasible, at least for non highly optimized kernels. Following the proposed performance model, the starting point is set at the coordinates designated by the measured peak with micro-benchmarks (top black point). Afterwards, the performance point is adjusted twice (E_{mix}, E_{instr}) by applying the efficiency degradations (E_{mix} and E_{instr}) and the attained compute performance drops by almost an order of magnitude. Therefore, the estimated compute performance is not expected to exceed 203 GFLOPS. The overall performance is estimated again by following the horizontal dashed line to the left till it crosses the kernel's line. In this case the kernel would be estimated to execute in 5.17 msecs. The actual measured performance of the kernel on the GTX-660 is designated by the "X" on the kernel's half-line (169.11 GFLOPS) and it corresponds to an execution time of 6.20 msecs.

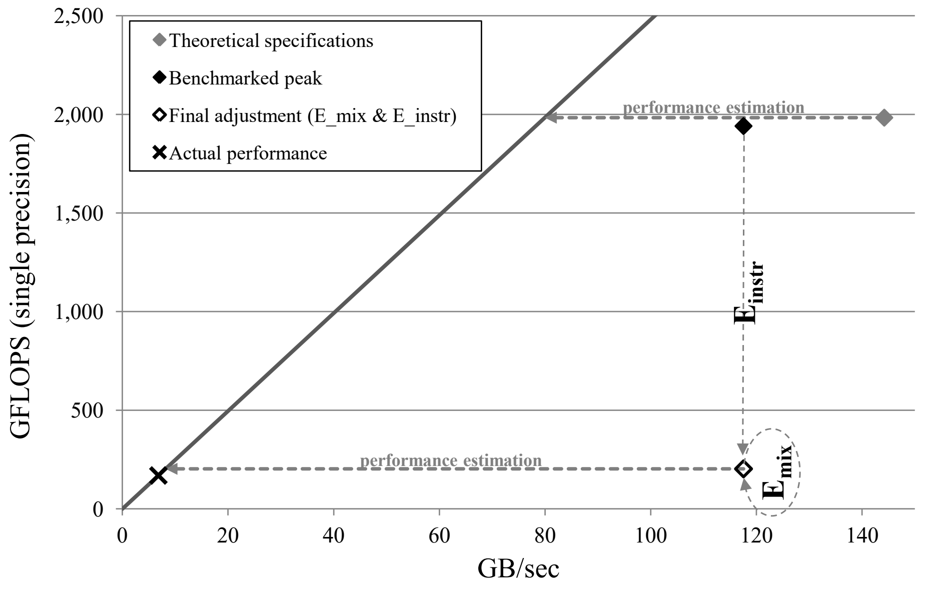


Figure 4.6: Visualization of SGEMM computation performance estimation on GTX-660 with efficiency adjustments versus using theoretical specifications.

It is evident that the proposed estimation time is significantly closer to the actual measured execution time compared to a pure theoretical peak approach. And this was mostly attributed to the consideration of the load/store instructions in the performance model.

In order to further validate the performance limiting factor characterization, a study of the hardware utilization metrics was conducted. All hardware metrics provided by the nvprof tool named with the *"_utilization"* suffix represent a rough measurement of the utilization ratio of a particular GPU resource, ranging from "Idle(0)" to "Max(10)" (an exception is the *issue_slot_utilization* which provides a percentage of the issue slots that issued one or more instructions and is therefore irrelevant to this investigation). As such the utilization rates for the GTX-660 GPU were collected as shown in table 4.11. The highest utilizations potentially expose the resources that are mostly utilized. These resources constitute most

Table 4.11: Utilization metric values of the SGEMM kernel, on the GTX-660.

*Metric	Utilization description	Value
l1_shared_utilization	L1/Shared Memory	High (9)
ldst_fu_utilization	Load/Store Function Unit	High (8)
alu_fu_utilization	Arithmetic Function Unit	Low (2)
cf_fu_utilization	Control-Flow Function Unit	Low (1)
dram_utilization	Device Memory	Low (1)
l2_utilization	L2 Cache	Low (1)
sysmem_utilization	System Memory	Low (1)
tex_fu_utilization	Texture Function Unit	Idle (0)
tex_utilization	Texture Cache	Idle (0)

* The "_utilization" suffix from metric names has been omitted

likely the performance bottleneck of the kernel. In this case the L1 and shared memory function unit utilization is *High (9)* which means that the shared memory already works near its full potential. The load/store unit utilization is also almost equally high (*High (8)*) which was also expected due to the density of load/store instructions in the instruction stream. As the proposed model combines both the compute instructions and the load/store instructions in a single abstract pipeline these rates point towards having a compute bound kernel on the GTX-660 and thus, they are consistent with the outcome of the proposed performance model.

This example is a particular one that exhibits the merit of load/store operation consideration in the performance model. In this case the load/store instructions are the most frequent as they consist almost 50%. To exhibit the value of special consideration of load-/store operations we conducted the performance prediction procedure in a more simplified approach in which all load/store instructions are handled the same way as other instructions. In this case the prediction yielded an effective peak performance of 505 GFLOPS and 2.08 msecs execution time, which is far more optimistic that the 5.17 msecs predicted by the proposed approach. The load/store operations can play a dramatic role in performance and thus, it is important for a performance model to take into special consideration of these operations.

4.5 Performance model assumptions

The basic principle of the roofline approach suggests that either the peak compute throughput or the memory bandwidth is feasible. As such in the proposed performance model it is assumed that the kernels under consideration are either compute or memory bound instead of latency bound. However, it should be noted that the predicted performance is still useful on latency bound kernels from the perspective of an upper performance bound (or a lower execution time bound). On such cases the kernel programmer should focus on optimizing the kernel by eliminating the latency bottleneck, if possible. The amount of parallelism must be able to keep the GPU computational units highly fed, as well. Additionally, all data are assumed to reside in the GPU memory, thus CPU-GPU transfer implications are not considered.

A summary of the assumptions that have been made in order to simplify performance prediction is provided below:

- Kernel performance is only bounded by instruction throughput or memory bandwidth, i.e. implying a compute or memory bound kernel. Other latencies are out of scope of the work conducted for this thesis. However, it should be noted that the predicted performance serves as an upper performance limit on latency bound kernels, which is still useful information from the perspective of providing an upper bound on performance (or a lower bound from the execution time perspective). On such cases the kernel programmer can be guided by considering the performance model's feedback on focusing to optimizing the kernel by eliminating the latency bottleneck, if possible.

- All instructions are executed by a single type of pipeline, thus mixing different types of instructions is not expected to improve IPC performance compared to applying a single type of instructions. Thus, multi-issuing instructions on different types of execution units is not considered apart from the primary execution unit and the memory subsystem of the GPU, e.g. mixing single precision floating point and load/store instructions.

- From a throughput standpoint all instructions are considered to belong to either of 3 categories:

  - Compute instructions
    They can be either single/double precision floating point or integer instructions. Peak throughput is determined by considering multiply-add operations. Typically, a multiply plus an addition operation are combined to a single multiply-add instruction.

  - Load/Store instructions
    Instructions involving loads and stores from various memory types. Peak throughput is determined by considering shared memory load/store operations, which tend to be the most efficiently implemented.

  - Control/overhead instructions
    All other instructions not included in the first two categories. Peak throughput is determined by considering integer addition instructions.

- Caching behavior tends to be relatively similar between different types of GPUs. Thus, the amount of DRAM transactions that is profiled on the reference GPU is not expected to change dramatically on the rest of the GPUs. This assumption is based on the fact that GPU caches can mostly exploit of spatial locality instead of temporal locality. It is believed that memory accesses with spacial locality are more predictable and the various GPU architectures exhibit more often a uniform behavior, while exploiting this type of locality.

- PCI-Express transfer overhead is out of scope of this thesis. All data are assumed to reside in the GPU memory, thus CPU-GPU transfer implications are not considered.

The goal of all of the assumptions applied on the performance model is twofold. First, a high level performance model should include a significant amount of details in the form of parameters in order to provide reasonably accurate results. Otherwise, its merit would be compromised. Secondly, it should not include an excessive amount of input as redundant details would not be significant to the performance estimation and they would unnecessarily complicate the process without returning significant benefits. There is also a third reason for not including some desired parameters which is the lack of hardware profiling data as provided by the GPU vendor. This limitation will be further discussed in a later chapter (7.2).

5. EXPERIMENTAL EVALUATION

In this chapter the performance estimation procedure is applied on a wide set of real world GPU kernels. These include a stencil computation kernel which is considered as a memory bound computation and a single precision matrix multiplication (SGEMM) kernel which is considered as compute bound. As a last and broader evaluation study, the performance estimation was tested on a large kernel subset of the well known Rodinia GPU benchmark suite[18]. The stencil computation and the matrix multiplication kernels have been developed for the purpose of the work summarized in this thesis. The prediction results are compared and discussed with the real execution results. Special cases where the predictions diverge significantly from real measurements are further investigated.

Table 5.1: List of the CUDA GPUs used in the experiments

	Architectural specifications				
GPU	C.C.	GPU architecture	# SMs	# SPs	SP clock rate
	(Compute Capability)	(Name)	(Total count)	(Total count)	(As reported by CUDA)
GTX-480	2.0	Fermi	15	480	1,550 MHz
GTX-660	3.0	Kepler	5	960	1,097 MHz
GTX-960	5.2	Maxwell	8	1,024	1,329 MHz
GTX-1060 6GB	6.1	Pascal	10	1,280	1,708 MHz
Tesla M2050	2.0	Fermi	14	448	1,147 MHz
Tesla K20c	3.5	Kepler	13	2,496	705 MHz

A list of all GPUs that have been used in the experiments is provided in table 5.1. First all micro-benchmarks were executed on all GPUs in order to derive the required parameters for the model. The derived parameter values are shown in table 5.2. These parameters are used for all the experiments on this thesis.

Table 5.2: *GPU parameters* as measured with micro-benchmarks

	Floating point ops		Integer ops		Memory access ops	
	T_{SP}	T_{DP}	T_{int}	T_{add}	B_{mem}	T_{ldst}
GPU	(GFLOPS)	(GFLOPS)	(GIOPS)	(GIOPS)	(GB/sec)	(GOPS)
GTX-480	1,462.20	184.09	742.34	732.86	163.36	369.73
GTX-660	1,940.80	89.70	359.04	621.36	117.56	169.58
GTX-960	2,842.70	89.67	955.37	1,426.15	86.35	295.64
GTX-1060 6GB	4,609.54	145.02	1,533.61	2,304.10	161.64	524.27
Tesla M2050	1,011.36	508.91	513.10	504.88	*107.44	255.68
Tesla K20c	3,115.24	1,153.08	584.26	969.28	*151.72	283.59

* with ECC enabled

The derived weight factors are provided in table 5.3. As previously explained, these weights represent a comparison of throughput in each operation with respect to the throughput in single precision operation execution. As an example, the Tesla Kepler GPUs feature double precision operation throughput at a rate of ⅓ of its single precision operation throughput. This is approximated by weight $W_{op\ (fp64)} = 2.70$ (Tesla K20c) which means that the measured throughput of single precision operations was 2.70 times higher than the measured throughput in double precision operations. Similarly, $W_{ldst} \approx 2$ (Tesla M2050) on Fermi architecture due to the amount of 16 load/store elements per 32 SPs in each multiprocessor. On Kepler architecture the number of load/store elements is 32 per 192 SPs in each multiprocessor hence W_{ldst} approaches to 6 (GTX-660 & Tesla K20c). This information will be used for the performance evaluation for all experiments.

Table 5.3: The GPU cost weights as measured and used in the model

	Instruction throughput weights				
GPU	W_{op} (fp32)	W_{op} (fp64)	W_{op} (int)	W_{ldst} (load/store)	W_{other} (add)
GTX-480	1.00	7.94	1.97	1.98	1.00
GTX-660	1.00	21.64	5.41	5.72	1.56
GTX-960	1.00	31.70	2.98	4.81	1.00
GTX-1060 6GB	1.00	31.79	3.00	4.40	1.00
Tesla M2050	1.00	1.99	1.97	1.98	1.00
Tesla K20c	1.00	2.70	5.33	5.49	1.61

As a reference GPU the GTX-480 is being used for all the experiments in this chapter. The computer systems used for the experiments were running 64bit Linux OS and the installed CUDA versions were v6.5, v7.0 and v7.5 for Tesla K20c, Tesla M2050 and all GTX cards, respectively. The GTX GPU systems were running on Intel Core i5 2500 CPUs, on Ubuntu 14.04.4 OS with kernel v4.2.0. The Tesla M2050 GPU system was equipped with dual socketed Intel Xeon X5650 CPUs, running a Debian based distribution, whereas the Tesla K20c GPU system was equipped with an Intel Core i7-3970X CPU, on Ubuntu 14.04 with kernel v3.13.0. Shared memory configuration was set to default (48KB for Fermi and Kepler), except for *Hotspot3D* of the Rodinia suite where a 16KB shared memory configuration was identified after inspection of the source code. Setting does not affect Maxwell and Pascal architectures, both of which utilize dedicated shared memories.

5.1 Applied kernel experiments

As said, the executed experiments include two variants of a stencil computation, a matrix multiplication (SGEMM) kernel and a large subset of the Rodinia benchmark suite[18]. Though, the stencil computation is traditionally considered as memory intensive and the matrix multiplication as compute intensive, these assumptions were verified by the proposed performance prediction method. Findings were also verified by probing on the appropriate utilization profiling metrics.

Beyond the aforementioned kernels the prediction model was also applied on the *mixbench* micro-benchmark itself, the results of which are presented in section 5.2.4. This is provided as a proof of the concept on a set of ideal kernels with various operational intensities.

5.1.1 Red/black SOR stencil computation

One of the first experiments that had been developed in the endeavor on the accomplishment of this thesis was the experimentation and analysis of the red/black SOR stencil computation[44, 45]. Specific methods for stencil computations provide large scale of parallelism and can be therefore applied on GPUs. However, the design and implementation is still a subject for potential optimizations.

In this section the stencil computation method is presented as well as some important implementation details. In case the reader is not interested in the implementation details he is suggested to skip this section.

Method description

The SOR (Successive Over-Relaxation) method belongs to the family of iterative methods like the Jacobi method which are widely used for solving large PDE (Partial Differential Equation) problems. In order for this method to perform well on the GPU it should be adequately parallelized. In this regard, the red/black ordering of the elements was chosen which allows straightforward parallelization. More specifically, within each iteration half of the elements can be calculated independently of the others, since there is no data dependence between them. For a 2D mesh of points an example is illustrated on figure 5.1. Points outside the dashed rectangle define the boundary conditions. Therefore, the problem in this form is ideal for parallelization.

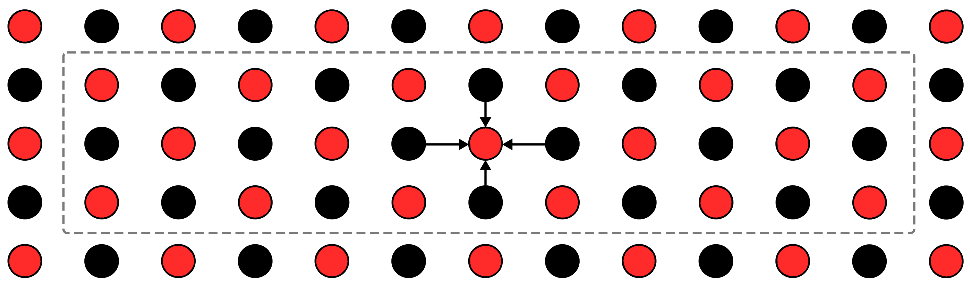


Figure 5.1: Same colored points depend only on the adjacent opposite colored points.

In this implementation the Laplace equation (5.1) is considered in 2D space (2 independent variables) as depicted below:

$$\nabla^2 T = 0 \quad or \quad \frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} = 0 \tag{5.1}$$

In order to solve the PDE, a predefined rectangular domain is discretized yielding a finite number of grid points. Inner grid points are the unknowns and the boundary point values are predefined as Dirichlet boundary conditions are assumed.

All these points are allocated in a 2D array during the calculation. The calculation is performed iteratively, in two phases. First, all red elements get updated and then all black elements follow. Every point is updated according to the neighbor point values, as equations (5.2) and (5.3) indicate.

$$u_{i,j}^{k+1} = (1-\omega)u_{i,j}^{k} + \omega(u_{i-1,j}^{k} + u_{i+1,j}^{k} + u_{i,j-1}^{k} + u_{i,j+1}^{k})/4 \quad \text{for } (i+j) \text{ odd} \tag{5.2}$$

$$u_{i,j}^{k+1} = (1-\omega)u_{i,j}^{k} + \omega(u_{i-1,j}^{k+1} + u_{i+1,j}^{k+1} + u_{i,j-1}^{k+1} + u_{i,j+1}^{k+1})/4 \quad \text{for } (i+j) \text{ even} \tag{5.3}$$

where ω is the relaxation factor which enables faster convergence in the computation. The speed of convergence is not examined in this paper, thus the selected ω value is irrelevant to our analysis. For the needs of our experiments a couple of a thousand iterations was selected as they seemed to be a reasonable choice. In practice the number of iterations is always related to the selected mesh size in order to satisfy convergence. Larger meshes require significantly more iterations to be performed till they converge.

The memory wall challenge: Stencil computation performance analysis

An important characteristic of this application is the particular degree of operational intensity with respect to GPUs capabilities. As it will be shown the red/black stencil computation is typically a memory bound application. Before applying the proposed method for extracting the kernel parameters and estimating the operation intensity in an automated fashion, a theoretical analysis will be provided. Thereafter, the theoretical and experimental estimations shall be compared.

First, it is required to estimate the mean number of accesses required for each computed element. This estimate can indicate a lower bound for global memory traffic requirements. On each iteration, during the first phase of computation all red and black elements are read and the red element values are written back. On the next phase, all element values are read again and the black element values are written back. Thus, assuming that either L1 or L2 cache is being used efficiently, N^2 access reads and $N^2/2$ access writes per red or black calculation stage are required, where N is the size of matrix in each dimension.

This equals to $3xN^2$ total accesses per full iteration, as (5.4) indicates.

$$\underbrace{N^2 + N^2/2}_{\text{red calculation phase accesses}} + \underbrace{N^2 + N^2/2}_{\text{black calculation phase accesses}} = \underbrace{3N^2}_{\text{total accesses per iteration}} \tag{5.4}$$

For each element about 6 floating point computations are required (equations (5.2) and (5.3)). So, for the whole matrix $6xN^2$ floating point computations and $3xN^2$ memory transfers of floating point values are required, thus the mean number of floating point computations per memory transfer of floating point value ratio is 2, or $1/4$ floating point computations per byte memory transfer ratio as double precision floating point computations are used. Comparing this ratio to the capability ratio of a GPU proves that the ratio is significantly lower. For instance, according to the theoretical specifications of the GTX-480 GPU (table 3.1) the GPU provides about 168 GFLOPS peak computational power in double precision operations and about 177 GB/sec memory bandwidth, which entails a ratio of about 0.95 Flops per byte. Thus, about 4x compute operations would be required in order to completely hide the memory access time with computations. Consequently, the performance is predominantly dependent on the memory bandwidth capability of the GPU device.

Implementation details

Some details regarding the applied implementation are provided here. The computation consists of a large number of iterations where each comprises two steps.

A straightforward approach would keep the original data arrangement of the matrix as provided. Applying computations on this arrangement however entails some memory access inefficiencies. Essentially, the applied memory store operations are strided as in each phase either red elements or black elements are stored. This results in memory write transactions that contain interleaved unmodified elements. Thus, the memory accesses would have to be uncoalesced to a significant degree as only half of the elements that are accessed within a memory store transaction are actually updated in each stage of computation. Consequently, the memory bandwidth is not optimally used. Considering the fact that the kernel is memory bound, a more optimized memory pattern has to used.

In order to avoid sparse element accesses and improve coalescing, a better approach would be reordering the matrix elements according to its color. The matrix is split into two independent matrices, one holding the red elements and the other the black elements (figure 5.2). Each element position (i, j) is transformed to a new one $(i/2, j)$ on the new matrix which is the red one if $(i + j) \mod 2 = 0$ or the black one otherwise. Therefore, the addressing of elements is now slightly more complex. The neighbor elements are positioned as 3 vertical elements plus one on the left or right, depending on the row number (figure 5.2: i, ii).

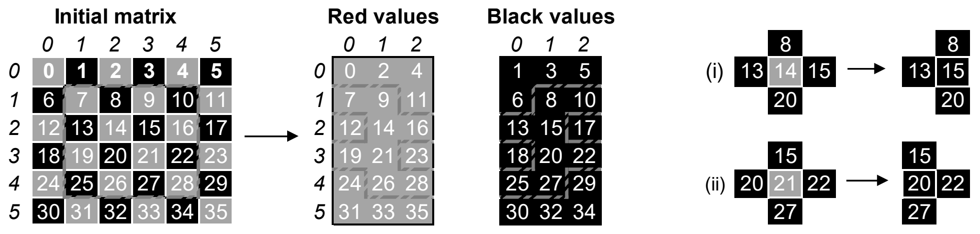


Figure 5.2: Positioning of elements in the reordered matrix by applying reordering by color.

In the new data arrangement all calculated elements in a row are contiguous and stored without leaving untouched elements between them. Thus, memory store transactions carry only updated element values and thus, the average efficiency is increased.

Threads of a thread block share data element values through shared memory and thus, all global memory read accesses are performed in a coalesced manner. One exception is reading the values for the overhanging halo elements (figure 5.3). Light gray elements are read in a coalesced manner whereas dark gray elements are read individually without coalescing, since they belong to different memory segments, which is inefficient but unavoidable. In this case, the global memory caching serves as a mechanism to avoid fetching the same data from global memory multiple times.

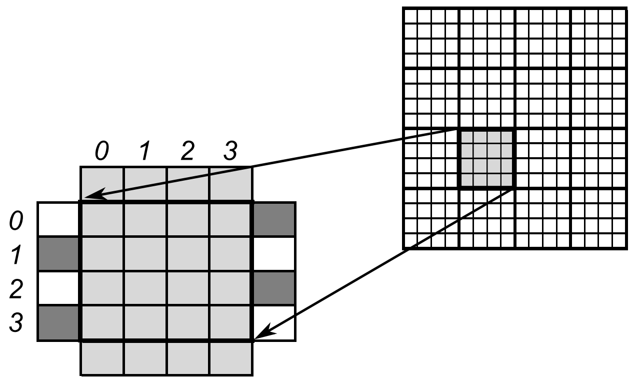


Figure 5.3: Read accesses by a hypothetical 4x4 thread block.

The implementation performs double precision calculation. Template variables were used instead of kernel parameters in order to fine tune the compiled kernel code. The data reordering procedure, which is the initialization and finalization of the reordered data, induces a small overhead for large matrices. However, as larger matrices require more iterations for computation and each iteration benefits from the reordered memory structure, the induced overhead becomes only a small fraction of the total computation.

Readers are referred for more information on the published work[44, 45] describing the respective implementations and experiments.

5.1.2 LMSOR stencil computation

A similar work had also been conducted regarding a relevant stencil computation method called LMSOR (Local Modified SOR)[20, 21, 22]. This method resembles the red/black SOR and has similar performance characteristics. However, in the particular case there is an opportunity to reduce memory traffic by decreasing the redundancy of utilized data. This decrement though, comes at a cost of re-computations which increase the compute requirements. This increment combined with the decrement in memory traffic requirements disturbs the operational intensity of the kernel.

In this section a kernel variation was chosen which makes use of a higher operational intensity value compared to the same value of the red/black SOR kernel and it theoretically poses a stronger candidate in exhibiting ideal operational intensity.

Method description

A brief mathematical background on the LMSOR method is provided in this paragraph. Reader is suggested to omit this section in order to proceed to implementation details.

This method was introduced by Boukas and Missirlis in [14] and the idea is based on letting the relaxation factor ω vary from equation to equation, as well as having two different sets of parameters $\omega_{1ij}, \omega_{2ij}$ to be used for the red ($i + j$ even) and black ($i + j$ odd) points, respectively. This means that each equation has its own relaxation parameter denoted by ω_{1ij} or ω_{2ij}, depending on the point color.

In particular, the solution of the second order convection diffusion equation is considered:

$$\Delta u - f(x, y)\frac{\partial u}{\partial x} - g(x, y)\frac{\partial u}{\partial y} = 0 \tag{5.5}$$

on a domain $\Omega = \{(x, y)\}|0 \le x \le 1,\ 0 \le y \le 1\}$, where $u = u(x, y)$ is prescribed on the boundary $\partial\Omega$. The discretization of (5.5) on a rectangular grid $M_1 \times M_2 = N$ unknowns within Ω leads to

$$\begin{aligned} u_{ij} &= \ell_{ij}u_{i-1,j} + r_{ij}u_{i+1,j} + t_{ij}u_{i,j+1} + b_{ij}u_{i,j-1}, \\ & \quad i = 1, 2, \ldots, M_1\ ,\ j = 1, 2, \ldots, M_2 \end{aligned} \tag{5.6}$$

with

$$\begin{aligned} \ell_{ij} &= \frac{k^2}{2(k^2 + h^2)}(1 + \frac{1}{2}hf_{ij})\ ,\ r_{ij} = \frac{k^2}{2(k^2 + h^2)}(1 - \frac{1}{2}hf_{ij}) \\ t_{ij} &= \frac{h^2}{2(k^2 + h^2)}(1 - \frac{1}{2}kg_{ij})\ ,\ b_{ij} = \frac{h^2}{2(k^2 + h^2)}(1 + \frac{1}{2}kg_{ij}), \end{aligned} \tag{5.7}$$

where $h = 1/(M_1 + 1)$, $k = 1/(M_2 + 1)$, $f_{ij} = f(ih, jk)$ and $g_{ij} = g(ih, jk)$. For a particular ordering of the grid points (5.6) yield a large, sparse, linear system of equations of order N of the form:

$$Au = b. \tag{5.8}$$

A grid point (i, j) is considered as red when $i + j$ is even and as black when $i + j$ is odd. The LMSOR method can be expressed as follows:

$$\begin{aligned} u_{ij}^{(n+1)} &= (1 - \omega_{1ij})u_{ij}^{(n)} + \omega_{1ij}J_{ij}u_{ij}^{(n)},\ \text{for}\ \ i + j\ \ \text{even} \\ u_{ij}^{(n+1)} &= (1 - \omega_{2ij})u_{ij}^{(n)} + \omega_{2ij}J_{ij}u_{ij}^{(n+1)},\ \ \text{for}\ \ i + j\ \ \text{odd} \end{aligned} \tag{5.9}$$

where

$$J_{ij}u_{ij}^{(n)} = l_{ij}u_{i-1,j}^{(n)} + r_{ij}u_{i+1,j}^{(n)} + t_{ij}u_{i,j+1}^{(n)} + b_{ij}u_{i,j-1}^{(n)} \tag{5.10}$$

and J_{ij} is called the local Jacobi operator. The parameters $\omega_{1ij}, \omega_{2ij}$ are called local relaxation parameters and (5.9) is referred to as the local Modified SOR (LMSOR) method [14]. In case the eigenvalues μ_{ij} of the local Jacobi operator J_{ij} are all real or all imaginary Boukas and Missirlis [14] found the optimum values of the local relaxation parameters ω_{1ij} and ω_{2ij} for the LMSOR method.

In the related published work[20, 21, 22] one optimization strategy that has been applied was the redundant computations. Re-computations can potentially be beneficial in cases where memory accessing becomes a bottleneck, i.e. memory bound kernels. Instead of keeping the processing units idle, one strategy is to recompute data when applicable, in order to avoid multiple memory accesses. This is a trade-off and in cases when a kernel is bandwidth limited, compute resources can be traded for less demand in memory bandwidth. It can be applied when a few operations at most are required for the computation, so that re-computation does not turn itself into a bottleneck. It can provide a performance speed-up and moreover, it can alleviate memory requirements and consequently allow solving larger problems. Moreover, even in cases where re-computation is excessively applied, although performance is worsened, there can be other benefits as it leaves more memory available for use and thus, a bigger problem is can be effectively solved.

In the previous work[20, 21, 22] a total of three kernel variations had been developed. Each variation applies re-computations to a different degree and thus each one is characterized by a different operational intensity value.

More specifically a summary of the kernels developed follows:

GPU Kernel #1 - No re-computations. This kernel performs no re-computations and from this stand point it is a straightforward implementation. It utilizes a total of six matrices in GPU memory and thus, it performs 8 element accesses per computed element. The ratio of floating point operations per byte accessed is theoretically estimated to be 0.17 ($^{11}/_{8\times 8}$), which is particularly low.

GPU Kernel #2 - Re-computations of elements $\boldsymbol{\ell_{ij}}$, $\boldsymbol{r_{ij}}$, $\boldsymbol{t_{ij}}$, $\boldsymbol{b_{ij}}$. On this kernel the number of matrices and the number of memory accesses are both decreased by 2. However, it comes at the cost of extra operations needed to recompute the required terms for the formula for each element on every iteration. In this case, each element computation requires 6 accesses and at least 15 floating point operations, as formulae (5.6) and (5.11) indicate. Now, the operational intensity is estimated to be 0.31 ($^{15}/_{6\times 8}$) flops per byte, which is roughly double than the same ratio of the straightforward implementation.

GPU Kernel #3 - Re-computations of elements $\boldsymbol{\ell_{ij}}$, $\boldsymbol{r_{ij}}$, $\boldsymbol{t_{ij}}$, $\boldsymbol{b_{ij}}$, $\boldsymbol{\omega_{ij}}$. In addition to the previous re-computations on this kernel the ω_{ij} term are also recomputed. Thus, in this case 5 accesses per computed element are required. However, a rough estimate is that at least 39 (15+24) flops are required. An approximation of the previous ratio is ~0.98 ($^{39}/_{5\times 8}$). Thus, in theory this kernel seems to be adequately balanced, as opposed to the previous kernels. During computation u_{ij}, f_{ij}, g_{ij} terms are accessed from memory and all other terms are recomputed as required.

Though kernel #3 was the theoretically optimum one, experiments proved that kernel #2 is actually the best performing[20, 21]. Therefore, the latter was chosen for performance analysis in this thesis. Kernel is #2 applies re-computations for the estimation of ℓ_{ij}, r_{ij}, t_{ij}, b_{ij}, by exploiting the values f'_{ij} and g'_{ij}. The latter are precomputed and stored in two matrices, f' and g', which have been defined as $f'_{ij} = \frac{1}{2}hf_{ij}$ and $g'_{ij} = \frac{1}{2}kg_{ij}$. Thus, 4 matrices are replaced by 2 matrices in device memory. The required terms (i.e. ℓ_{ij}, r_{ij}, t_{ij} and b_{ij}) are recomputed on demand through f'_{ij} and g'_{ij} during the LMSOR iterations as follows:

$$\ell_{ij} = w(1 + f'_{ij}) \ , \ r_{ij} = w(1 - f'_{ij})$$
$$t_{ij} = w(1 - g'_{ij}) \ , \ b_{ij} = w(1 + g'_{ij}), \tag{5.11}$$

where $w = \frac{h^2}{2(k^2+h^2)}$, which is constant during the computation.

For additional information on the implementations and analysis, readers are referred to the aforementioned published work[20, 21, 22].

5.1.3 Matrix multiplication (SGEMM)

As a second experiment a kernel performing single precision matrix multiplication was selected. The implementation applies tiling of source matrices in shared memory in order to minimize global memory traffic as an implicit method for caching of source matrix element tiles. Each thread block is allocated an equally sized tile for each source matrix in shared memory. Thus, the size of the thread block also defines the size of shared memory allocated for each source matrix. Each thread in a thread block fetches one element for each source matrix to shared memory. Since the size of the thread block is known during compilation time, the compiler is able to fully unroll the inner loop of computation. Due to the dense computations, this kernel is considered as a compute intensive kernel.

5.1.4 Rodinia benchmark suite

As a last and broader application experiment of the performance model, the Rodinia benchmark suite version 3.1 was selected[18]. Its source code is freely available on the internet and it has been previously used by the research community. Rodinia consists of a large set of benchmarks in CUDA, OpenCL and OpenMP parallel programming environments. The CUDA implementation is comprised of 23 total benchmarks, each of which involves one or more CUDA kernels. The list of the Rodinia CUDA benchmarks is provided in table 5.4.

Build configuration notes

For practical reasons some changes had to be applied in both the source code and build configuration files. The purpose was to allow execution and profiling on the whole range of hardware used for the experiments, as a wide range of architectures is used. Additionally, some peculiarities in the source code forced the kernels exhibiting abnormal performance behavior and therefore, they were fixed. The list of the changes/corrections applied is provided below in order to allow the reproduction of the experimental results of the same benchmark environment:

Table 5.4: CUDA benchmark list of the Rodinia benchmark suite

Benchmark name / folder	Domain
Back Propagation / backprop	Pattern Recognition
CFD Solver1 / cfd	Fluid Dynamics
Heart Wall / heartwall	Medical Imaging
Huffman / huffman	Lossless data compression
LavaMD2 / lavaMD	Molecular Dynamics
MUMmerGPU / mummergpu	Bioinformatics
Needleman-Wunsch / nw	Bioinformatics
Breadth-First Search1 / bfs	Graph Algorithms
GPUDWT / dwt2d	Image/Video Compression
HotSpot / hotspot	Physics Simulation
Hybrid Sort / hybridsort	Sorting Algorithms
Leukocyte / leukocyte	Medical Imaging
Myocyte / myocyte	Biological Simulation
Particle Filter / particlefilter	Medical Imaging
SRAD / srad	Image Processing
B+ Tree / b+tree	Search
Gaussian Elimination / gaussian	Linear Algebra
Hotspot3D / hotspot3D	Physics Simulation
Kmeans / kmeans	Data Mining
LU Decomposition / lud	Linear Algebra
k-Nearest Neighbors / nn	Data Mining
PathFinder / pathfinder	Grid Traversal
Streamcluster1 / streamcluster	Data Mining

- All CUDA code was compiled using the Fermi code generation option (*-gencode= arch=compute_20,code=\"compute_20,sm_20\"*) in order to allow execution on all GPUs used in the experiments and disabled L1 caching option (*"-Xptxas -dlcm=cg"*) in order to minimize cache dependent behavior. On the original configuration files the target GPU hardware was set on a per benchmark basis. This compiler option also guides the code to be compiled in a PTX GPU form. Code in PTX form is compiled to ISA code in a JIT fashion during runtime.

- In two benchmarks (Hotspot3D and Huffman) the debugging flags had been used (-g -G) which produced fairy slow executable code and therefore they were removed.

- Floating point literal suffixes ("f") were added in source code where required. On five benchmarks (backprop, HotSpot, Leukocyte, Myocyte and SRAD) it was observed that some floating point literals in the kernel codes had been declared by using the double precision notation, i.e. missing the "f" suffix, whereas the corresponding data types were declared as single precision types. This induced an inadvertent implicit conversion of the operand to double precision type which in turn led the compiler to produce double precision operations where single precision were clearly intended.

This induced a severe negative impact on performance and therefore, it was corrected.

- Reduced the number of iterations on CFD Solver1 from 2000 to 20 and set attempts to 1 ("#define ATTEMPTS 1") on Myocyte in order to allow the profiling to succeed. Otherwise, profiling counters were led to overflow and the profiling procedure failed.

Considerations and performance estimation

After profiling the execution of kernels of the Rodinia suite it was observed that a significant subset of them did not provide configurations with adequate parallelism and therefore the GPU occupancy was significantly low. Consequently, these kernels are expected to exhibit performance limitations due to limited occupancy of the GPUs which require vast amounts of parallelism. Thread parallelism is expressed in terms of thread block counts and sizes of thread blocks. Therefore, on the purpose of this analysis lower bound limits were set on the minimum accepted thread block size and thread block count. In order for kernels to qualify to performance analysis, the invocations have to be configured with at least 64 sized thread blocks and a minimum of 90 thread blocks in total.

This decision has been taken in order to exclude latency related cases where kernel performance is not dependent on the overall GPU throughput but on other factors instead. For instance, the completion time of individual thread blocks tends to be more dependent on the GPU clock frequency. In general the excluded kernels are considered as latency bound. However, the use of predicted execution times on them serves as an upper limit on the attained performance. This analysis though, is restricted to the kernels meeting those limits. The kernels that were selected are provided in table 5.5. Out of the 55 total kernels only 28 qualify. As such, no kernel has been used in the experiments from *Heart Wall*, *Needleman-Wunsch*, *Myocyte*, *Particle Filter*, *Gaussian Elimination*, *LU Decomposition* and *Streamcluster1* benchmarks of Rodinia which didn't meet the configuration requirements. From this point and on, the abbreviations will be used in order to refer to the particular kernels.

5.2 Performance prediction experiments

In this section all produced results are analyzed and presented, along with the real measurements. Results include experiments on all aforementioned applications, i.e. red/black SOR, LMSOR, SGEMM and Rodinia kernels. Comparisons of real and predicted measurements are discussed.

Table 5.5: List of selected kernels of the Rodinia suite used on the experiments

Benchmark	Kernel name	Abbreviation
Hotspot3D	hotspotOpt1	3d-htsp
B+ Tree	findK	btr-fnd
	findRangeK	btr-rng
Back Propagation	bpnn_adjust_weights_cuda	bp-adj
	bpnn_layerforward_CUDA	bp-fwd
Breadth-First Search1	Kernel	bfs-k1
	Kernel2	bfs-k2
GPUDWT	c_CopySrcToComponents<int>	dwt-cpy
	dwt_cuda::fdwt53Kernel<int=192, int=8>	dwt-krn
CFD Solver1	cuda_compute_flux	e3d-flux
	cuda_compute_step_factor	e3d-sfac
	cuda_initialize_variables	e3d-init
	cuda_time_step	e3d-step
HotSpot	calculate_temp	hspt-tmp
Hybrid Sort	mergepack	hs-pack
	mergeSortFirst	hs-srtf
Kmeans	invert_mapping	km-map
	kmeansPoint	km-pt
LavaMD2	kernel_gpu_cuda	lvmd-krn
Leukocyte	dilate_kernel	lct-dil
	GICOV_kernel	lct-gic
MUMmerGPU	mummergpuKernel	mum-krn
	printKernel	mum-prt
k-Nearest Neighbors	euclid	nn-euc
PathFinder	dynproc_kernel	pfnd-krn
Huffman	vlc_encode_kernel_sm64huff	pvl-huff
SRAD	srad_cuda_1	srad-c1
	srad_cuda_2	srad-c2

5.2.1 Red/black SOR stencil computation

In the previous chapter the performance model was applied on the red/black SOR stencil computation kernel[44, 45] and the performance analysis was provided for the GTX-660 GPU. In this section the analysis is conducted for all the rest of the CUDA GPUs on the same implementation and problem configuration. As previously, the computation is conducted on a 8192x8192 2D array u comprised of double precision values and using predefined boundary conditions. The reordering by color has been applied in order to enforce locality and coalescing[44, 45]. Shared memory has been utilized for reusing values of the mesh by intra-block threads and its configuration has been set to default, which is 48KB shared memory for Fermi and Kepler platforms. Maxwell and Pascal architectures utilize a dedicated 96KB of shared memory per SM on consumer GPUs[73, 75].

The captured kernel metrics and the derived kernel parameters have already been provided in tables 4.4 and 4.5, respectively. As it was estimated the kernel operational intensity is $O_{krn} = \frac{W_{comp}}{W_{traf}} \approx 0.3$. Recall from the previous section that the theoretically determined ratio was $^1/_4$ which is reasonably close.

The rest of this section will be focused on the performance estimation on the rest of the GPUs. The relative execution cost per instruction type is estimated by using equations (4.13), (4.14) and (4.15) with the weight factors already provided in table 5.3. The derived values and E_{instr} for all GPUs are provided in table 5.6. The relative execution cost is a measure expressing the relative contribution of each instruction type to the whole execution time, in case the kernel is compute bound. The E_{instr} efficiency rate expresses an approximation of the relative instruction execution cost of all beneficial compute instructions (double precision in this case) in the whole instruction stream.

Table 5.6: The relative instruction execution costs and the instruction efficiency on all GPUs for the red/black SOR kernel.

GPU	C_{op}	C_{ldst}	C_{other}	E_{instr}
GTX-480	0.97	0.33	0.71	48.09%
GTX-660	2.63	0.97	1.11	55.89%
GTX-960	3.85	0.81	0.71	71.72%
GTX-1060	3.86	0.74	0.71	71.72%
Tesla M2050	0.24	0.33	0.71	10.83%
Tesla K20c	0.33	0.93	1.14	13.70%

After applying the efficiency factors (E_{mix} and E_{instr}) to get the adjusted throughput using equation (4.17) the T'_{op} can be derived. This suggests the maximum expected performance of this kernel, ignoring the required DRAM bandwidth. The comparison of O_{dev} and O_{krn} determines if the kernel is expected to be compute of memory bound. The expected performance is estimated by (4.18). The performance estimation intermediate values and results are given in table 5.7. It is evident that the kernel is expected to be memory bound with the exception of GTX-660, on which it was compute bound as it was proven in the previous chapter. On the GTX-480 is marginally characterized as memory bound.

Table 5.7: The derivation of performance estimation on all GPUs for the red/black SOR kernel (O_{krn}=0.3).

GPU	T_{op}	T'_{op}	B_{mem}	O_{dev}	$O_{krn} > O_{dev}$	$T_{predicted}$
	(GFLOPS)	(GFLOPS)	(GB/sec)	(ops/byte)	(True:Compute bound)	(GFLOPS)
GTX-480	184.09	51.07	163.36	0.31	False	49.31
GTX-660	89.70	28.92	117.56	0.25	True	28.92
GTX-960	89.67	37.10	86.35	0.43	False	26.07
GTX-1060 6GB	145.02	60.80	161.64	0.38	False	48.79
Tesla M2050	508.91	55.12	107.44	0.51	False	32.43
Tesla K20c	1153.08	91.13	151.72	0.60	False	45.80

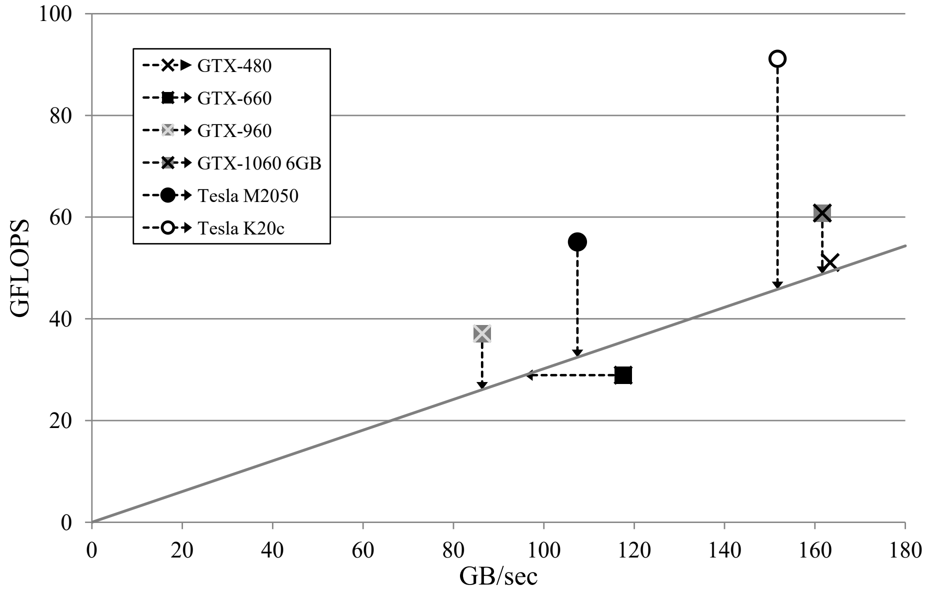


Figure 5.4: The proposed model applied on all GPUs for the red/black SOR computation.

In figure 5.4 the prediction results after the efficiency adjustments have been applied are illustrated. The solid dark gray half-line represents the red/black kernel's operational intensity (slope ≈ 0.3 flops/byte). The points refer to the GPUs expected performance on the particular kernel after all the adjustments have already been applied. It is apparent that the only GPU point residing below the kernel's half-line is the one representing the GTX-660. Additionally, the point for GTX-480 is very close to the half-line.

Similarly to the case study in previous chapter we provide profiling measurements of the utilization metrics. These are helpful in the validation process of the results. As such we collected the utilization rates for all GPUs in table 5.8. The highest utilizations potentially expose the resources that are mostly utilized. Beyond the GTX-660 GPU, the rest GPUs exhibit the DRAM utilization as the highest rated metric exposing the memory bound limitation. As such, the utilization metrics are consistent with the outcome of the proposed performance model.

Table 5.8: The highest rated utilization metric values of the red/black SOR kernel, on all GPUs.

GPU	*Metric/utilization (1)	*Metric/utilization (2)	*Metric/utilization (3)
GTX-480	dram/High (9)	l2/High (7)	alu_fu/Mid (5)
GTX-660	alu_fu/High (9)	dram/High (7)	l1_shared/Mid (4)
GTX-960	dram/High (8)	double_precision_fu/Mid (6)	l2/Mid (6)
GTX-1060 6GB	dram/High (9)	double_precision_fu/Mid (6)	l2/Low (3)
Tesla M2050	dram/High (9)	l2/Mid (6)	l1_shared/Mid (4)
Tesla K20c	dram/High (9)	ldst_fu/Mid (5)	l2/Mid (4)

* The "_utilization" suffix from metric names has been omitted

In table 5.9 the predicted and measured execution times are provided with the respective

Table 5.9: Prediction results on all GPUs for the red/black SOR stencil computation

GPU	Predicted time (msecs)	Measured time (msecs)	Error (%)
GTX-480	20.414	21.456	-4.86%
GTX-660	34.803	34.851	-0.14%
GTX-960	38.620	38.793	-0.45%
GTX-1060 6GB	20.632	20.994	-1.73%
Tesla M2050	31.038	33.367	-6.98%
Tesla K20c	21.979	23.482	-6.40%

prediction error. The error percentage is evaluated by the ratio expressed in (5.12).

$$Error = \frac{T_{Predicted} - T_{Measured}}{T_{Measured}} \times 100\% \tag{5.12}$$

All GPUs exhibited less than 7% error. Tesla GPUs exhibited a slightly larger error which can be attributed to the ECC implications on these platforms. Of course, the ECC cost has already been indirectly accounted in the performance model through micro-benchmarking. However, this kernel employs uncoalesced memory accesses for the left and right halo point elements within a tile and the cost of ECC can be increased on these cases. Red/black SOR kernel is memory bound on Tesla GPUs and thus the additional overhead of ECC memory has impact on the kernel's execution time.

5.2.2 LMSOR stencil computation

The computation involves memory accesses on matrices of size 3842x3842 with double precision values. Similarly to the red/black kernel, reordering by color has been applied and shared memory configuration has been set to default (48KB shared memory for Fermi and Kepler platforms). This kernel utilizes texture memory for implicit caching of data and alleviating global memory traffic. The profiling metrics were captured for a total of 4 iterations of computation on the GTX-480, using the *nvprof* tool for all iterations of the red elements' computation as shown in table 5.10.

Afterwards, the kernel parameters are determined and depicted in table 5.11 along with the operational intensity. The theoretical operational intensity was estimated to be 0.31, which compared to the experimental estimation ($O_{krn} = 0.46$) is significantly lower. This difference is justified by the actual amount compute operations performed. Using the theoretical assumptions as provided on paragraph 5.1.2, for matrix sizes of 3842x3842 and 4 iterations of red elements computation, the expected amount of compute operations would be 442,368,000, while the expected memory traffic would be 1,415,577,600 bytes. The experimental memory traffic estimation is very close to $W_{traf} = 1{,}463{,}296{,}768$, but the compute operation count, $W_{comp} = 679{,}312{,}384$, is significantly higher than the theoretically estimated value (more than +53%). This entails 23 flops per element on average instead

Table 5.10: The profiling metrics gathered for the LMSOR kernel on a GTX-480 GPU. All metrics are the accumulated values collected in 4 iterations for the red element calculation.

Parameter	Measurement
M_{fma32}	0
M_{fma64}	36,864,000
M_{ldst}	93,107,518
M_{inst}	18,433,804
M_{fp32}	0
M_{fp64}	132,964,096
M_{int}	184,601,469
$M_{tran\text{-}r}$	9,577,528
$M_{tran\text{-}w}$	1,854,478

Table 5.11: The *kernel parameters* and the operational intensity of the LMSOR kernel.

Parameter	Value
K_{type}	fp64
W_{comp}	679,312,384
W_{traf}	1,463,296,768
E_{mix}	63.86%
D_{ops}	22.54%
D_{ldst}	15.78%
D_{other}	61.68%
O_{krn}	0.46

of the assumed 15 flops in paragraph 5.1.2. Nevertheless, the operational intensity is still low and this kernel is also expected to be memory bound. It still exhibits a large instruction overhead $D_{other} = 61.68$%, though lower than the exhibited ratio of the red/black SOR kernel.

The performance modeling process proposes that this kernel is compute bound on the GTX-660 GPU and memory bound on the other GPUs, similarly to the red/black SOR kernel. The performance prediction results are provided in table 5.12 and compared with the actual execution measurements. The GTX-480 and GTX-960 GPUs exhibited performance very close to the predicted one. The rest of the GPUs executed the kernel with up to ~10% worse performance than predicted. By inspecting the top utilizations seen on the GPUs, provided in table 5.13, it is obvious that the GTX-660 is clearly compute bound as the ALU unit is saturated to max(10) utilization, which validates the performance model classification. The slight performance prediction error could stem from the evaluation of the computational workload cost. The same kernel on the GTX-1060 seems to reside on the critical/turning point between compute and memory bound regions as the DRAM and ALU utilizations are equally assessed to be high(9). This is also expressed by the performance model which estimated a device compute to memory ratio (O_{dev}) equal to 0.479 which is very close to the operational intensity of the kernel (0.46). This would be expressed by having the kernel's operational intensity approaching the ridge point of

Table 5.12: Prediction results on all GPUs for the LMSOR kernel stencil computation

GPU	Predicted time (msecs)	Measured time (msecs)	Error (%)
GTX-480	8.957	8.971	-0.15%
GTX-660	16.397	18.069	-9.26%
GTX-960	16.946	17.458	-2.93%
GTX-1060 6GB	9.053	10.132	-10.65%
Tesla M2050	13.619	15.162	-10.17%
Tesla K20c	9.644	10.399	-7.26%

Table 5.13: The highest rated utilization metric values of the LMSOR SOR kernel.

GPU	*Metric/utilization (1)	*Metric/utilization (2)	*Metric/utilization (3)
GTX-480	dram/High (9)	alu_fu/High (7)	l2/High (7)
GTX-660	alu_fu/Max (10)	dram/High (7)	l2/Low (3)
GTX-960	dram/High (8)	double_precision_fu/High (7)	l2/Mid (6)
GTX-1060 6GB	dram/High (9)	double_precision_fu/High (9)	l2/Mid (4)
Tesla M2050	dram/Max (10)	l2/Mid (6)	tex/Low (3)
Tesla K20c	dram/High (9)	l2/Mid (4)	alu_fu/Low (2)

* The "_utilization" suffix from metric names has been omitted

the roofline performance line in the roofline model. The Tesla GPUs exhibited high(9) or max(10) DRAM utilizations so the prediction error could possibly be attributed to ECC causing an extra overhead.

5.2.3 Matrix multiplication (SGEMM)

For the matrix multiplication kernel the selected matrix sizes for the experiments were 1280x640 and 640x640. The program was compiled by setting the thread block size to be 32x32 (1024 threads per block). The larger the thread block size is set, the larger the shared memory tile and therefore, the less the produced memory traffic is expected to be.

The derived parameters for this kernel have already been shown in table 4.10. The operational intensity is particularly high in this case ($O_{krn} = 24.81$) so it would be valid to assume that this kernel is compute bound. However, using the peak compute throughput of the device as the expected kernel's compute performance would not be wise in a sufficiently accurate performance model. For instance, the GTX-660 with a theoretical 1,983 SP GFLOPS peak, would ideally execute this workload in less than 0.53 msecs, which is far from the measured 6.2 msecs. If the instruction densities are taken into account then it becomes apparent that this kernel executes mostly load/store instructions (almost half of total instructions) instead of compute operations. Furthermore, as typically load/store operations are more expensive than single precision flops, kernel's compute potential is greatly reduced. This reduction is modeled through the E_{instr} factor. If this factor is applied to the measured peak compute throughput for the GTX-660, the adjusted throughput becomes $T'_{op} = 202.8$ GFLOPS and entails the execution time of 5.171 msecs, which is far closer to the measured performance.

In table 5.14 the prediction results are compared with the measured times. The performance of both the GTX-960 and GTX-1060 appears to be very close to the predicted time. For the rest of the GPUs the performance prediction error ranges between 16% and 26%. After inspection and experimentation it was found that groups of 4 load instructions on shared memory regarding sequential addresses were combined to single 128bit loads (*LDS.128* instruction on Fermi). In the profiling process these instructions account as single, though the actual throughput of these is about $1/4$ to $1/2$ of the equivalent using 32bit loads, depending on the architecture. Consequently, the accounted relative weight of load/store instructions is significantly less than the actual one. This relation of throughput ratios is justified by the vendor's documentation regarding shared memory throughput[73], as well as, it has been validated on a published work assessing on-chip memories' throughput via micro-benchmarking[43]. The GTX-960 and GTX-1060 predictions were coincidentally very close to the actual execution time due to the effective overlapped execution of load/store and compute instructions.

Table 5.14: Prediction results for the matrix multiplication kernel

GPU	Predicted time (msecs)	Measured time (msecs)	Error (%)
GTX-480	2.987	4.033	-25.95%
GTX-660	5.171	6.201	-16.61%
GTX-960	2.973	2.938	1.20%
GTX-1060 6GB	1.705	1.694	0.64%
Tesla M2050	4.320	5.795	-25.45%
Tesla K20c	3.122	3.963	-21.24%

In table 5.15 the highest rated utilization metrics are provided per GPU for the SGEMM kernel. The particularly high values of *l1_shared_utilization* or *shared_utilization metrics* point to the shared memory resource as the bottleneck of this application which is more or less connected to the load/store utilization.

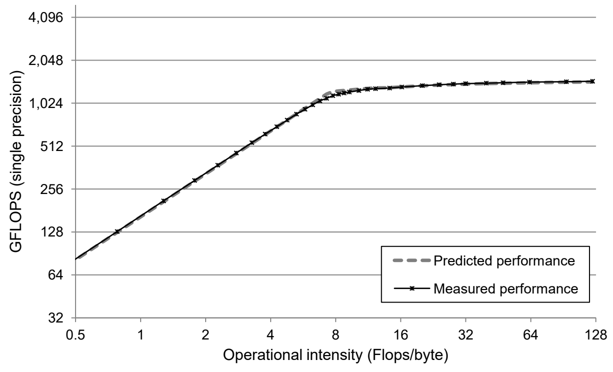
Table 5.15: Top GPU utilization metrics on matrix multiplication kernel profiling

GPU	Metric	Ratio
GTX-480	l1_shared_utilization	High (8)
GTX-660	l1_shared_utilization	High (9)
GTX-960	shared_utilization	High (9)
GTX-1060 6GB	shared_utilization	High (9)
Tesla M2050	l1_shared_utilization	High (8)
Tesla K20c	l1_shared_utilization	High (8)

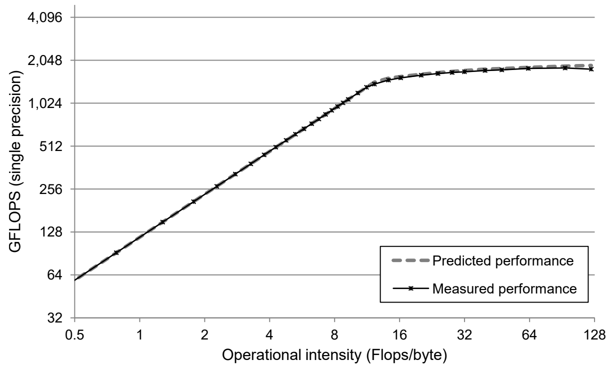
5.2.4 Mixbench performance prediction

As an additional evaluation of the proposed model the *mixbench* micro-benchmark was selected as a target. *Mixbench* is composed of a large group of kernels, where each one is differentiated by parameter values which determine the balance of compute/memory operations. Each parameter is essentially a template variable and each different value leads to compiling a different kernel binary in the executable. All kernels are optimized so the device is pushed close to its theoretical limits. Loops are optimally unrolled, proper data types are used, parallelism is abundant, no divergences, control instructions minimized, memory accesses are coalesced, etc. That said, the kernels are running very efficiently are presented in section 3.2.1.

In this section, the performance prediction model was applied on *mixbench* itself. The parameters of the kernels were captured and thereafter, the conducted performance prediction results were used to reconstruct the roofline chart. In addition, running the micro-benchmark on the same devices is used to compare the actual measurements with the predicted ones. The GTX-480 is used again as a reference GPU for profiling the kernel. This kind of chart is illustrated in figure 5.5 for the GTX-480 and GTX-660 GPUs. Figure 5.6 present the same information for the GTX-960 and GTX-1060, and figure 5.7 for Tesla M2050 and K20c, respectively.



(a) GTX-480

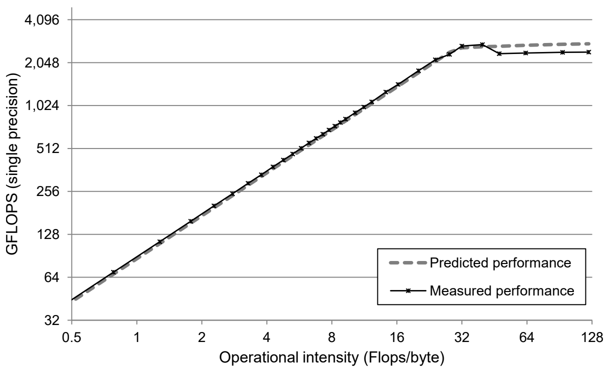


(b) GTX-660

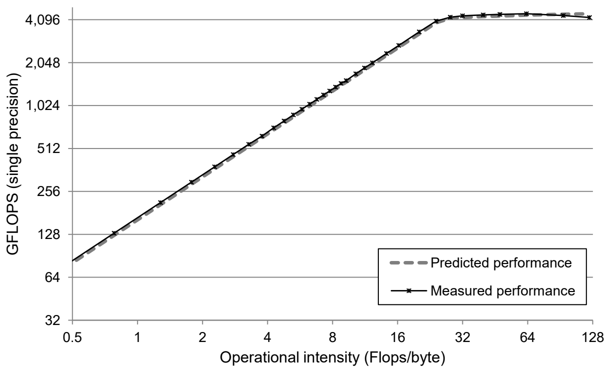
Figure 5.5: Performance prediction on mixbench (SP) for GTX-480 and GTX-660.

On the GTX-960 prediction there is a small divergence observed for high operational intensities. This prediction error exposes some different compilation behavior for this architecture. In order to elaborate, the same prediction procedure was repeated by using the GTX-960 GPU as a reference this time in order to investigate the prediction results by eliminating any code generation side effects. The prediction results are provided in figure 5.8. Here the regenerated prediction clearly exposes the performance fluctuation.

As it is evident, the model in general led to estimated performance which was quite close to the actual measured. In fact, the curvature of line is very accurately predicted by the proposed model compared to just using the flat theoretical peaks as already shown in figures 3.2, 3.3 and 3.4. Real world kernels, however, frequently expose other bottlenecks but on

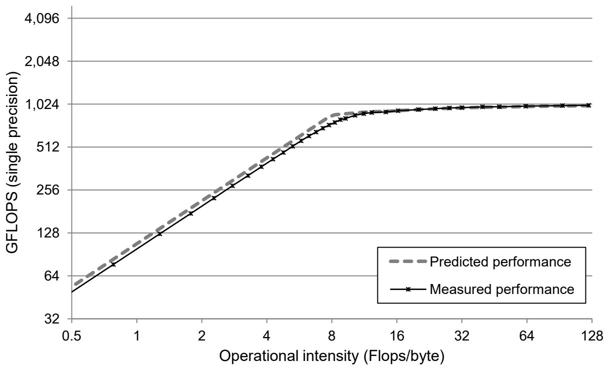


(a) GTX-960

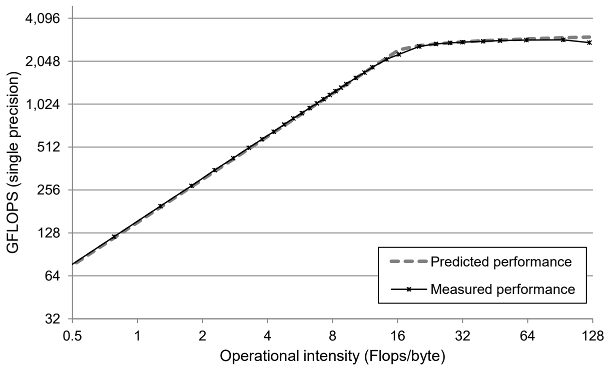


(b) GTX-1060

Figure 5.6: Performance prediction on mixbench (SP) for GTX-960 and GTX-1060.



(a) Tesla M2050



(b) Tesla K20c

Figure 5.7: Performance prediction on mixbench (SP) for the two Tesla GPUs.

these ideal kernels the proposed model produces close to real measurement predictions.

5.2.5 Rodinia benchmark suite

The Rodinia execution experiments were focused on the selected 28 kernels as provided in table 5.5. The experimental executions yielded the parameter values that are summarized in table 5.16. Out of 28 selected kernels exactly half of them were characterized as single precision floating point (fp32), 13 as integer (int) and just one as double precision floating point (fp64). Some are compute intensive e.g. in terms of instruction density on *e3d-flux* more than 44% of executed instructions are single precision floating point and on *lvmd-krn* more than 36% of executed instructions are double precision floating point. In terms of operational intensity *lct-gic* approaches 600 flops/byte which is particularly high. A detailed roofline chart with all single precision Rodinia kernels on GTX-480 is depicted in figure 5.9.

Using the aforementioned parameters the prediction results were estimated and exper-

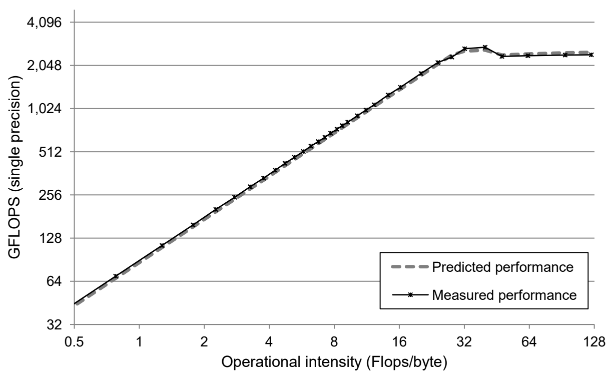


Figure 5.8: Performance prediction on mixbench (SP) for GTX-960 by using same GPU as reference.

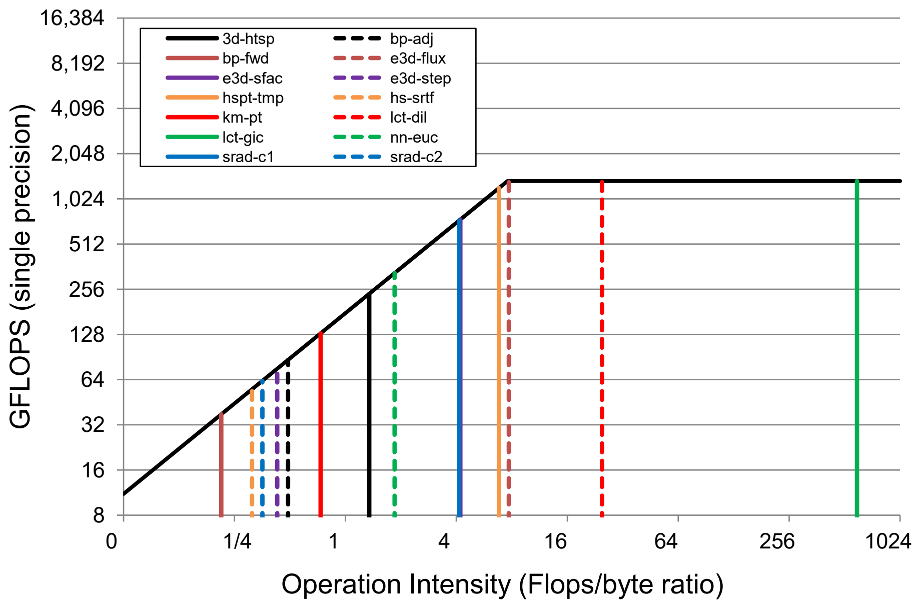


Figure 5.9: GTX-480 roofline using the estimated fp32 Rodinia kernel intensities.

imental executions were conducted in order to compare the former with the actual time measurements. The prediction errors are depicted in figure 5.11. Out of 168 total predictions for all GPUs, 30 predictions proved to be pessimistic as the respective errors were positive, which means that the actual execution times were shorter than predicted. The rest 138 predictions were optimistic and this is expressed with the negative prediction errors. In addition, 40 predictions were highly optimistic as the error was below -50%. In terms of APE (Absolute Percentage Error), which is the absolute error ($|Error|$), the majority of predictions on all GPUs ($^{89}/_{168} = 52.98$%) exhibited less than 25% APE. Focusing on each particular GPU, predictions falling below 25% APE were 53.6% ($^{15}/_{28}$), 53.6% ($^{15}/_{28}$), 57.1% ($^{16}/_{28}$), 50.0% ($^{14}/_{28}$), 50.0% ($^{14}/_{28}$) and 53.6% ($^{15}/_{28}$) for the GTX-480, GTX-660, GTX-960, GTX-1060, Tesla M2050 and Tesla K20c, respectively. Thus, in summary prediction accuracy is acceptable for more than half of predictions for all GPUs.

For sake of the evaluation of the efficiency factors (E_{mix} & E_{instr}) in figure 5.10 the pre-

Table 5.16: Collected *kernel parameters* and the operational intensity of the selected Rodinia kernels

Kernel	K_{type}	W_{comp}	W_{traf}	E_{mix}	D_{ops}	D_{ldst}	D_{other}	O_{krn}
3d-htsp	fp32	1,703,936,000	1,264,790,400	83.33%	15.92%	11.43%	72.65%	1.347
btr-fnd	int	138,477,928	12,264,768	50.00%	54.95%	9.57%	35.48%	11.291
btr-rng	int	149,868,966	14,210,080	50.00%	58.75%	11.34%	29.91%	10.547
bp-adj	fp32	9,437,296	19,332,000	64.29%	15.91%	20.45%	63.64%	0.488
bp-fwd	fp32	2,031,616	9,576,448	50.00%	1.51%	7.86%	90.63%	0.212
bfs-k1	int	144,663,996	419,167,488	50.00%	15.59%	3.95%	80.46%	0.345
bfs-k2	int	76,010,748	24,407,808	50.00%	35.13%	7.39%	57.48%	3.114
dwt-cpy	int	20,709,376	15,775,936	50.00%	54.86%	20.83%	24.31%	1.313
dwt-krn	int	61,932,984	26,166,336	50.00%	51.92%	25.87%	22.21%	2.367
e3d-flux	fp32	4,862,041,500	630,725,760	81.70%	44.11%	3.93%	51.96%	7.709
e3d-sfac	fp32	230,343,560	54,782,720	85.23%	34.35%	3.46%	62.20%	4.205
e3d-init	int	3,788,928	5,851,008	50.00%	44.83%	17.24%	37.93%	0.648
e3d-step	fp32	157,386,240	369,014,400	96.43%	16.28%	18.60%	65.12%	0.427
hspt-tmp	fp32	23,790,048	3,490,880	80.95%	12.93%	6.52%	80.55%	6.815
hs-pack	int	171,966,464	135,862,912	50.00%	43.08%	25.74%	31.17%	1.266
hs-srtf	fp32	41,958,520	134,636,032	50.00%	27.78%	5.56%	66.67%	0.312
km-map	int	129,438,028	611,707,072	50.00%	73.18%	18.99%	7.83%	0.212
km-pt	fp32	508,840,600	693,726,720	74.64%	29.56%	29.22%	41.22%	0.733
lvmd-krn	fp64	11,415,296,000	329,011,328	78.79%	36.07%	4.08%	59.86%	34.696
lct-dil	fp32	144,440,534	5,853,952	50.00%	9.62%	9.61%	80.76%	24.674
lct-gic	fp32	2,886,264,648	4,827,456	81.37%	36.27%	13.44%	50.29%	597.885
mum-krn	int	64,023,008	122,884,896	50.00%	24.67%	2.11%	73.22%	0.521
mum-prt	int	34,693,612	103,484,288	50.00%	25.78%	3.87%	70.35%	0.335
nn-euc	fp32	513,168	277,536	66.67%	27.23%	9.08%	63.70%	1.849
pfnd-krn	int	163,099,165	43,833,920	50.00%	41.43%	19.43%	39.14%	3.721
pvl-huff	int	401,590,480	14,557,120	50.00%	47.39%	9.09%	43.51%	27.587
srad-c1	fp32	847,249,408	204,695,744	81.45%	24.44%	8.70%	66.86%	4.139
srad-c2	fp32	83,886,080	236,833,856	83.33%	5.25%	16.66%	78.08%	0.354

diction error is provided for three prediction cases on the GTX-480 GPU. The first is not applying efficiency factor corrections at all, the second applies just E_{mix} correction and the last one applies the method as proposed by utilizing both E_{mix} and E_{instr}. It is evident that in all cases the efficiency adjustments either improve the absolute prediction error or in the worst case they have no affect.

According to figure 5.9, taking into account just the operational intensity of the kernel leads to expect a total of 11 fp32 kernels being memory bound on the GTX-480 GPU. Our analysis, though, exposed only 4 kernels (*bp-adj*, *e3d-step*, *hs-srtf*, *km-pt*) exhibiting higher intensity than the one the GPU could offer.

The errors of the rest of the predictions were larger and in some cases they reached to high levels, exceeding 70% APE. In this regard, table 5.17 with the highest utilizations on the execution of the Rodinia kernels is provided for all GPUs. The highest rated utilization metric potentially reveals the most congested resource of the GPU and therefore it is a significant indication of the performance limitation factor on the particular GPU. As the proposed model is based on the assumption that kernels are not latency bound, the most

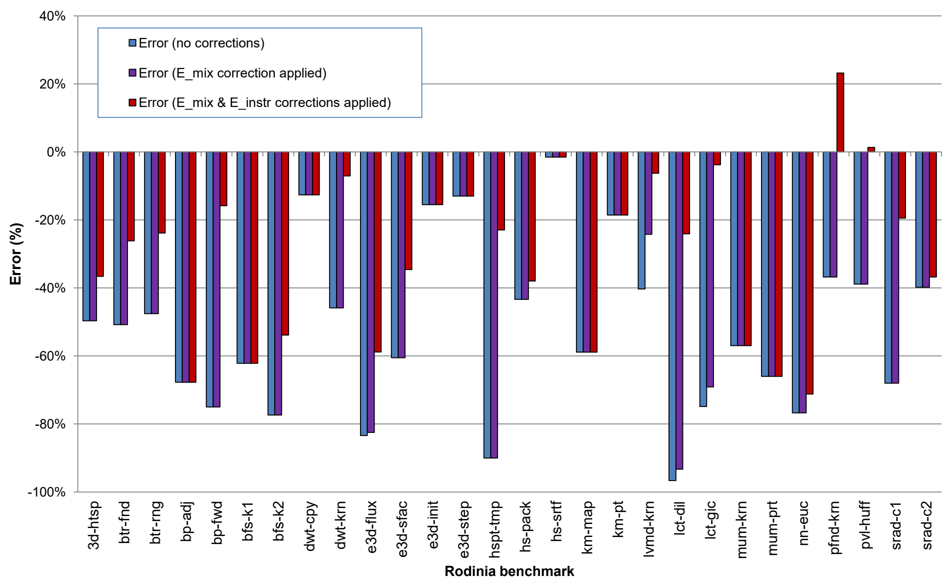


Figure 5.10: Prediction errors in the Rodinia suite kernels on GTX-480, in relation to the exploitation of the efficiency factors.

utilized resource should be adequately utilized (e.g. utilization $\geqslant 7$). Otherwise, the kernel should be considered as latency bound and the predicted execution time is expected to be significantly lower than the actual one. In this case it serves as a lower bound. In addition, the utilization metric should be either connected to the DRAM or to an execution resource of the SM (ALU, Load/Store unit), otherwise performance is expected to depend on a resource utilization (e.g. L2 cache) that it not explicitly considered in the proposed model or it is bound to unknown latencies. To this end, some typical cases which exhibited the largest average prediction errors were investigated. In addition, some distinctive cases were probed where different behavior between GPUs was observed concerning the execution time prediction and resource utilizations.

Special case considerations

Kernel *pfnd-krn* is essentially the only one that performed better on Fermi based GPUs than predicted. This is attributed to the fact that this kernel was classified as type *int* and as such its compute workload is based on the throughput of integer instructions, where each integer instruction's cost is accounted by the model as the cost of a Multiply-Add integer instruction. However, this cost is not the lowest one exhibited by integer instructions. For instance, the integer addition instruction throughput is double on Fermi GPUs[73]. The inspection of the assembler code produced for Fermi (SASS) showed that the inner loop consisted of 31 instructions out of which only 3 were of MAD type. Apart from the 6 shared memory instructions (LDS/STS) there where 10 other integer instructions (ISETP, IADD, etc) and 4 data movement instructions (MOV) which are expected to execute efficiently. As there is no deeper distinction between integer instruction types in the available profiling

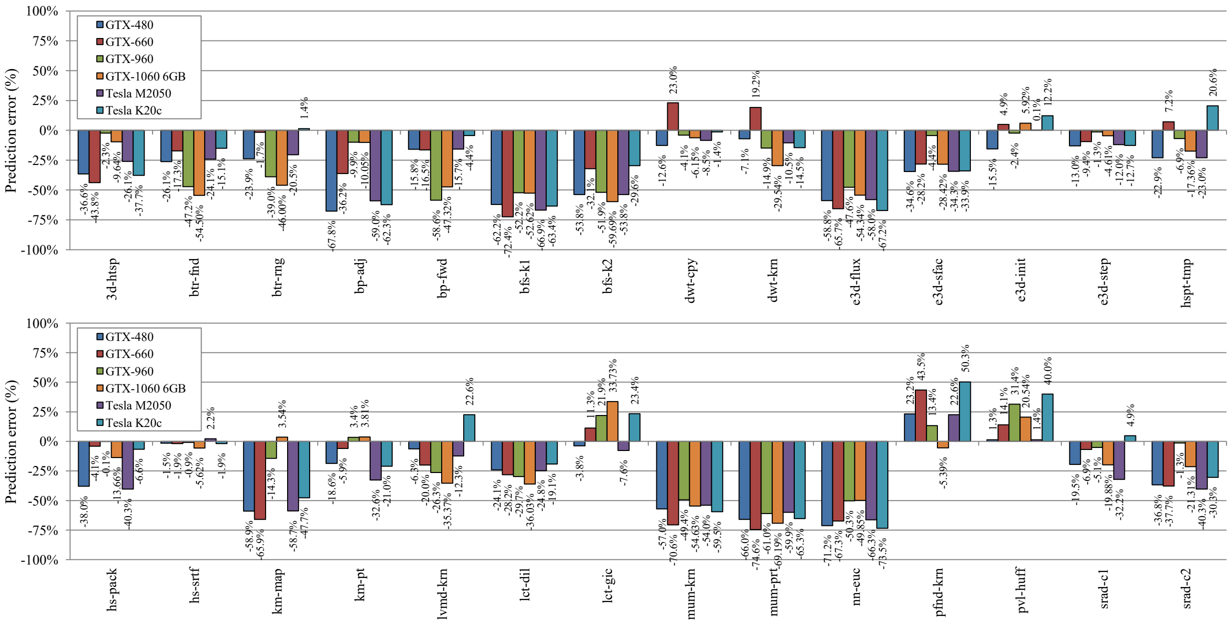


Figure 5.11: Prediction error of the selected Rodinia kernels per GPU.

metrics there cannot be an accurate cost assignment to integer instructions.

On Kepler GPUs the performance gap between the addition instruction throughput and the multiply-add instruction throughput is significantly higher as each SM is able to fulfill 160 addition instructions per clock compared to just 32 integer MAD instructions ($^1/_5$ throughput ratio)[73] and thus prediction error is wider on these GPUs. Maxwell and Pascal GPUs reach to a theoretical throughput of 128 integer addition instructions per clock ($^1/_4$ throughput ratio)[73], which entails also a wider gap compared to Fermi GPUs, though narrower to Kepler. In addition, all three recent GPU architectures (Kepler, Maxwell and Pascal) are able to co-issue ALU and load/store instructions contrary to Fermi GPUs. For both reasons, kernels *pvl-huff* and *lct-gic* executed faster than predicted on Kepler, Maxwell and Pascal GPUs.

Both *MUMmerGPU* kernels (*mum-prt* & *mum-krn*) exhibited large prediction errors, 66.00% and 57.51%, respectively. Especially, the prediction of the *mum-prt* kernel execution on the GTX-660 reached to 74.56%. All highest utilized metrics of both kernels point to the DRAM resource as seen in table 5.17. However, the highest profiling utilizations were observed particularly low for most GPUs (3-4) with Tesla GPUs being an exception, which indicates other latencies as bounds. After source code inspection, it was noticed that the inner loop global memory references consisted of 8bit scalar (*char*) type accesses. Experience has shown that GPUs cannot approach peak memory bandwidth by utilizing such short memory types. Ideally, both kernels would be memory bound as indicated of the proposed model. However, this inspection suggests that kernels are bound on the DRAM latency as the requested transactions cannot fully utilize the DRAM resource. In contrast, the Tesla GPUs exhibited higher rates on DRAM utilization due to the ECC protection, which caused a much larger DRAM traffic (3,840,153, 10,062,748 and 6,612,211

Table 5.17: The highest utilization metrics for the Rodinia kernels and the corresponding utilization rating.

Kernel	GTX-480	GTX-660	GTX-960	GTX-1060 6GB	Tesla M2050	Tesla K20c
3d-htsp	l2 (7)	alu_fu (4)	l2 (10)	dram (8)	l2 (8)	dram (6)
btr-fnd	alu_fu (3)	alu_fu (3)	l2 (5)	single_precision_fu (3)	alu_fu (4)	ldst_fu (4)
btr-rng	alu_fu (4)	ldst_fu (4)	l2 (6)	tex (4)	alu_fu (4)	ldst_fu (5)
bp-adj	l2 (4)	dram (5)	l2 (9)	dram (8)	dram (5)	dram (4)
bp-fwd	alu_fu (7)	alu_fu (7)	single_precision_fu (6)	single_precision_fu (5)	alu_fu (8)	alu_fu (7)
bfs-k1	alu_fu (2)	ldst_fu (2)	dram (3)	dram (2)	dram (4)	dram (3)
bfs-k2	alu_fu (4)	alu_fu (3)	dram (4)	dram (3)	alu_fu (4)	alu_fu (3)
dwt-cpy	dram (9)	dram (8)	dram (8)	dram (8)	dram (9)	dram (8)
dwt-krn	alu_fu (5)	ldst_fu (6)	dram (7)	dram (6)	dram (6)	dram (4)
e3d-flux	alu_fu (4)	alu_fu (3)	l2 (10)	dram (4)	alu_fu (4)	l2 (3)
e3d-sfac	alu_fu (6)	alu_fu (5)	dram (7)	dram (7)	alu_fu (6)	dram (6)
e3d-init	dram (9)	dram (8)	dram (7)	dram (8)	dram (9)	dram (8)
e3d-step	dram (8)	dram (8)	dram (8)	dram (8)	dram (9)	dram (7)
hspt-tmp	alu_fu (6)	alu_fu (6)	single_precision_fu (7)	single_precision_fu (7)	alu_fu (6)	alu_fu (5)
hs-pack	dram (8)	dram (6)	dram (8)	dram (8)	dram (10)	dram (8)
hs-srtf	dram (9)	dram (8)	dram (8)	dram (8)	dram (9)	dram (9)
km-map	ldst_fu (5)	ldst_fu (6)	dram (7)	dram (8)	dram (9)	ldst_fu (7)
km-pt	dram (8)	dram (8)	dram (8)	dram (9)	dram (10)	dram (8)
lvmd-krn	alu_fu (10)	alu_fu (10)	double_precision_fu (10)	double_precision_fu (10)	alu_fu (7)	alu_fu (8)
lct-dil	alu_fu (5)	alu_fu (3)	l2 (7)	single_precision_fu (4)	alu_fu (5)	alu_fu (5)
lct-gic	alu_fu (7)	alu_fu (5)	tex_fu (7)	tex_fu (7)	alu_fu (7)	alu_fu (5)
mum-krn	dram (4)	dram (3)	dram (4)	dram (3)	dram (8)	dram (6)
mum-prt	dram (4)	dram (3)	dram (4)	dram (3)	dram (8)	dram (5)
nn-euc	alu_fu (4)	dram (4)	l2 (7)	l2 (4)	l2 (5)	alu_fu (3)
pfnd-krn	alu_fu (5)	ldst_fu (6)	dram (7)	dram (6)	dram (6)	ldst_fu (6)
pvl-huff	alu_fu (5)	alu_fu (5)	shared (6)	shared (6)	alu_fu (5)	l1_shared (6)
srad-c1	alu_fu (6)	ldst_fu (6)	dram (8)	dram (7)	alu_fu (5)	dram (6)
srad-c2	dram (6)	ldst_fu (6)	dram (8)	dram (7)	dram (9)	dram (7)

∗The *"_utilization"* suffix in the metric names has been omitted for space conservation.

total transactions on GTX-480, Tesla M2050 and Tesla K20c, respectively). Therefore, this kernel is not expected to perform as optimally as the model would presume. As such, both kernels could not get to the performance levels that the proposed model predicted.

Another kernel exhibiting large prediction error is *nn-euc* (avg. APE 63.09%). However, this kernel takes a tiny amount of time to execute on the order of few micro-seconds (7.42 micro-seconds on GTX-480). Kernels executing on such small intervals are difficult to predict as the execution time becomes susceptible to the unavoidable invocation and execution overheads. Thus, short kernel invocations are not considered to utilize the GPU sufficiently enough in order to yield predictable execution times by using the proposed model.

Kernels *bfs-k1* and *bfs-k2* prediction results are expressed by a 61.63% and 46.82% APE, respectively. By inspecting the utilization metrics it becomes evident that the maximum utilization rate ranges between 2 and 4 which in this case is also quite low. The type of metric varies (alu_fu, dram, ldst_fu) but most important is the particularly low rate which indicates yet another latency bound kernel. A further inspection on the GTX-480 reveals that a total of 12 invocations were conducted per kernel with an average execution time of 0.56 msecs and 0.075 msecs, respectively. A significant amount of invocations took less than 0.1 msecs and as such, the invocation overheads are expected to affect the execution time severely. This fact justifies the observed errors.

The *“Back Propagation”* kernels exhibited some peculiar behavior on the GTX-960 and GTX-1060 performance predictions. While the *bp-adj* kernel’s performance was not well predicted in general (average error 40.87%), predictions on the GTX-960 and GTX-1060 GPUs were significantly more accurate with the respective APE at ~10%. In contrast, the performance of *bp-fwd* kernel was significantly better predicted on the rest of the GPUs, yet the GTX-960 and GTX-1060 in particular still exhibited different behaviors as the predicted time was about the half of the actual measured time. The DRAM utilizations of *bp-adj* on both the GTX-960 and GTX-1060 were significantly higher (7 and 8, respectively) than on the rest of the GPUs and thus, prediction was more accurate on these GPUs. The reason the DRAM was better utilized was again the reduced latencies on the particular GPUs. Stall reason metric values on the GTX-960 were *stall_memory_dependency*: 80.39% (which is normal for a memory bound kernel), *stall_exec_dependency*: 7.42% and *stall_sync*: 7.75%. In contrast, on the GTX-480 stall metrics ratios were *stall_memory_dependency*: 56.03%, *stall_sync*: 19.35%, *stall_exec_dependency*: 11.58% and *stall_pipe_busy*: 10.28%. Thus, it is evident that the GTX-480 is significantly more affected by the synchronization and pipeline latencies. On the other hand, during the inspection of *bp-fwd* kernel’s performance prediction it was found that the amount of compute operations on both the GTX-960 and GTX-1060 GPUs was significantly higher (>3) than the same amount on the other GPUs. By examining the kernel’s source code it was found that fast transcendental operations were involved in the computations (*__log2f()* and *__powf()*) which can be differently implemented on each GPU architecture, leading to different instruction counts. The experiments showed that the use of the transcendental operations increased the total amount of instructions significantly on different magnitudes between GPU architectures leading performance prediction errors to large margins. Nonetheless, it was observed that the use of these transcendentals could have been easily avoided by the programmer, as their operands were known during compilation time. Moreover, as the operands are essentially integers, their use on functions expecting floating point operand values is not expected to be optimal.

At this point, some cases are examined where kernels exhibited high utilizations on compute or memory resources but performance prediction accuracy was not equally high. In this respect, code inspection was used for better understanding and gaining insight into the possible causes for the inaccurate prediction. On a particular example, the *hs-pack* kernel on Tesla K20c exhibited “Max (10)” *dram_utilization* utilization while on the rest of the GPUs it was “High (8)”, and just “Mid (6)” on the GTX-660. However, Fermi GPUs performance prediction reached to error 38% and 40%, on the GTX-480 and Tesla M2050, respectively. It should be noted that the particular kernel execution is quite short as it takes less than 1.5 msecs to complete 4 total iterations which is 0.37 msecs per iteration. Hence, it is not surprising that a such short invocation exhibits up to a 40% error on some GPUs.

Kernel *lvmd-krn* was the only double precision compute (*fp64*) in the selected Rodinia kernels. All GTX GPUs saturated the double precision utilization to “Max (10)”. However, the average APE for the particular kernel was ~20.5%. A key indication lies on the *warp_execution_efficiency* profiling metric, which was 78.21% on GTX-480. This reflects the actual cause of the miss-predicted result, which was the use of conditional loops, the count of which was dependent on the thread index and thus, it caused divergent execution between threads of warps. Another observation is the fact that all GPUs exhibited lower performance due to the aforementioned cause with the exception of Tesla K20c, which exceeded the predicted performance by almost ~22.6%. The cause of this discrepancy on prediction errors is not clear and a further analysis will be required which is left for future investigation.

Few kernels provided apparently better prediction performance on the GTX-960 and GTX-1060 than on the rest of the GPUs. The APE on *km-map*, *3d-htsp* and *e3d-sfac* kernels was 14.32%, 2.31% and 4.36% on GTX-960, while the average APE was 41.52%, 26.03% and 27.31%, respectively. The same kernels exhibited an APE of 3.54%, 9.64% and 28.42% on the GTX-1060, respectively. A possible cause could be the more efficient L2 cache evident on both the GTX-960 and GTX-1060 which aids DRAM traffic reduction. However, this explanation requires a further investigation which is left as future work. Note that kernel *e3d-sfac* is called for a total of 20 times with an average execution time of just 0.042 msecs per invocation on the GTX-480 thus, its prediction is expected to be less accurate.

Last, kernels *dwt-cpy* and *dtw-krn* performed significantly better on GTX-660 GPU than predicted. The key element is that these kernels are integer compute based instead of floating point and therefore they are more susceptible to the integer instruction throughput variance. Using the proposed model they were designated as compute bound kernels on GTX-660 but the actual average cost of integer instructions in this kernel is less than presumed by the model and therefore the actual execution times are lower. In particular, the utilization metrics showed that *dwt-cpy* kernel is actually memory bound (table 5.17). The same issue holds for the K20c GPU as well, but as both performance limiting factors (compute / memory traffic) were more balanced on this particular GPU the effective prediction error was lower.

Exploitation of reference GPU execution time

In order to increase the accuracy of predictions as a fine tuning improvement, the observed kernel's execution time on the reference GPU can taken into account. As the GPU kernel is actually executed on the reference GPU during the profiling process, the observed execution time can be used as a measure of the effectiveness of execution when compared with the predicted time on the same GPU. In this respect, the *utilization factor* (E_{util}) is defined as given by (5.13):

$$E_{util} = \frac{T_{Measured}^{(refGPU)}}{T_{Predicted}^{(refGPU)}} \tag{5.13}$$

This factor can be applied as an additional correction on the predicted execution time of all the rest of the GPUs as $T'_{Predicted} = E_{util} \times T_{Predicted}$. However, it should be noted that this can be only a statistical correction as the actual performance limiting factor is not necessarily the same for all GPUs and it does not always affect performance to the same extent. Therefore, this correction is proposed to be only optionally applied.

After the *utilization factor* correction was applied on the previously presented Rodinia results, the adjusted prediction errors are depicted in figure 5.12. In this case the percentage of total predictions below 25% increased to $^{92}/_{140} = 65.71$%. The prediction results on the reference GPU were excluded, as the adjusted prediction error on the reference GPU is 0% by definition due to the correction applied to itself. However, some predictions have worsened and especially 5 particular predictions (*bp-adj* and *km-map* kernels) exhibited positive errors at the range of ~100% or more. The excessively high positive errors are attributed to the fact that on these cases the adjusted predicted time got significantly longer compared to the actual execution time. Due to the form of equation (5.12) the long predicted times tend to be reflected with particularly high error percentages. On the original predictions the error was mostly negative expressing the longer measured time compared to predicted one. Consequently, this correction discards the qualification of using the model to estimate a lower bound on the execution time.

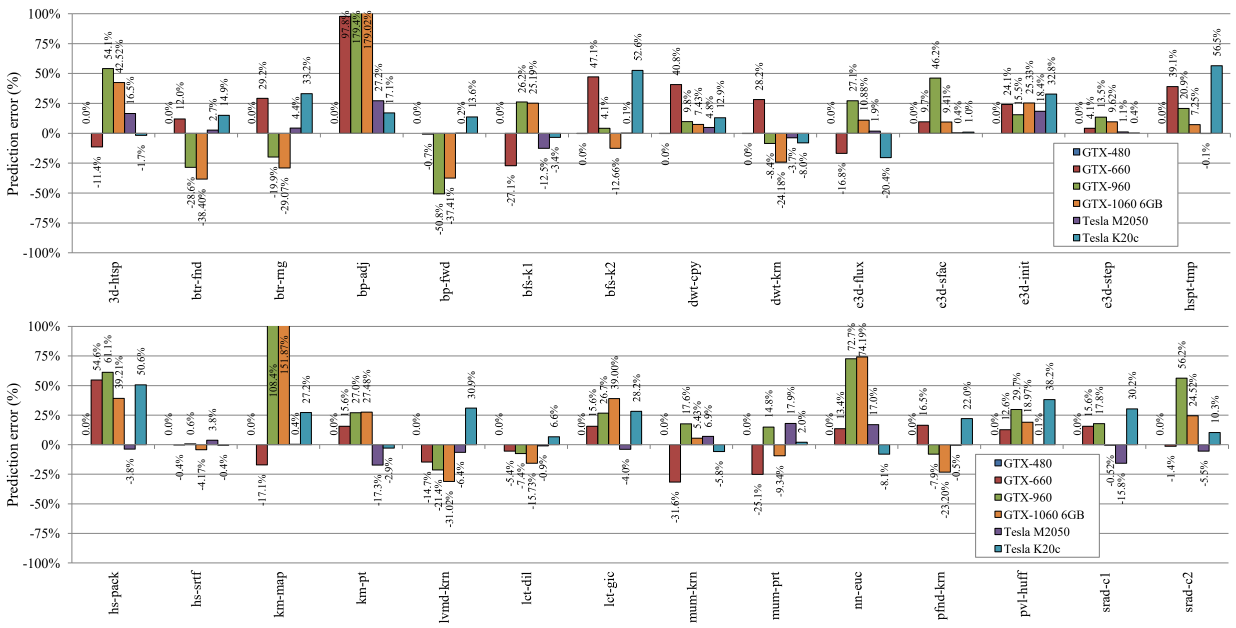


Figure 5.12: Rodinia prediction errors after applying *utilization factor* correction.

5.2.6 Summary and conclusions

By summarizing all the prediction results it is concluded that out of all conducted experiments about half of them exhibited less than 25% APE. This is considered a significant achievement given the little set of input that is used by the method. However, some cases exhibited relatively larger errors which exposed limitations characterizing the method. Additionally, some specific notes based on observations regarding the prediction results are provided in the following list:

- Stencil computations performed mostly as expected, being memory bound with the GTX-660 GPU slightly diverging to this trend. On both stencil computations the particular GPU exhibited compute bound behavior. In addition, the GTX-1060 on the LMSOR kernel was at the edge of being memory bound, having both the memory and ALU utilizations rated as High(9). LMSOR, in general, exhibited slightly larger errors presumably due to the slightly more complex memory access patterns as each computation requires fetching data from four matrices instead of one as it is the case with red/black SOR kernel.
- The SGEMM kernel exhibited larger errors, reaching up to ~26% APE on the GTX-480. The prediction error stems from the fact that the compiler generated 128bit shared memory accesses by combining groups of 32bit accesses on successive addresses. The 128bit load/store instructions take significantly longer to execute than their 32bit counterparts. The cost of load/store instruction execution, as perceived by the model, does not take into account the width of the data being accessed which is considered as a weakness of the proposed method.
- Some Rodinia kernels (e.g. *pfnd-krn*, *pvl-huff*, *lct-gic*, *dwt-cpy* and *dtw-krn*) exhibited higher performance than anticipated on one or more GPUs by applying the performance model. This is justified by the variability of integer instruction cost. This cost is accounted in the proposed model as the cost of a Multiply-Add integer instruction. There are multiple types of integer instructions which vary in performance as their instruction throughput depends on the underlying GPU architecture. Consider the integer throughput information as provided by the vendor in table 5.18. The variance in various types of integer operations can be very wide. Especially Kepler GPUs exhibit a $5\times$ variance (e.g. integer addition vs integer multiplication). When an integer computation kernel is comprised of faster instructions than multiply-add instructions then the kernel's performance can be higher than predicted by the proposed model.
- A less relevant issue that potentially causes significant disturbance on the performance results is the variance on GPU core frequencies during execution. This effect is more common on the more recent GPUs, and it was observed in the experiments on both Maxwell and Pascal GPUs, which at the time of writing this thesis were the most recent GPU CUDA architectures.
- It was also observed that memory accesses via small data types do not perform as predicted, e.g. 8bit types in *MUMmerGPU* kernels. This effect has to be investigated further and it is left as future work. It cannot be exposed through the profiling method employed by the proposed approach and as such it can be practically regarded as a latency bound case.
- Short kernel invocations (e.g. *nn-euc*, *hs-pack*) cannot be accurately predicted as the invocation overhead is significant. As a criterion of a minimum duration it is proposed to measure at least half a millisecond of execution time per invocation on the

Table 5.18: Throughput of native integer arithmetic instructions[73] (operations per clock cycle per multiprocessor)

Operation	2.0	2.1	3.0, 3.2	3.5, 3.7	5.0, 5.2, 5.3
32-bit integer add, extended-precision add, subtract, extended-precision subtract	32	48	160	160	128
32-bit integer multiply, multiply-add, extended-precision multiply-add	16	16	32	32	multiple instructions[1]
32-bit integer shift	16	16	32	64[2]	64
compare, minimum, maximum	32	48	160	160	64
32-bit integer bit reverse, bit field extract/insert	16	16	32	32	64
32-bit bitwise AND, OR, XOR	32	48	160	160	128
count of leading zeros, most significant nonsign bit	16	16	32	32	64

[1]experimentally estimated as $42.59 \approx {}^{128}\!/_{3}$ according to tables 5.2 and 5.3
[2]32 for GeForce GPUs

reference GPU. Otherwise, the prediction can be quite optimistic as no overheads are considered.

- Irregular computations, e.g. transcendental operations, cannot be directly supported by the performance method. These operation counts are not currently provided by the hardware profilers. This effectively means that the black-box approach of the method cannot work currently on the proposed method since more detailed information is required from the profiler. A discussion on this issue follows on the next chapter (section 6.2).

- Purely based on the utilization metrics, in few kernels the L2 cache seems to pose a bottleneck. On 13 cases out of the 186, it appears that the *l2_utilization* is the highest profiled utilization metric. Out of these, only 7 (<4%) exhibited a high utilization degree (≥ 7, e.g. High(10) on *e3d-flux* with GTX-960). These examples could potentially point to a hot-spot on caches regarding GPU performance. However, the L1 & L2 caches do not exhibit the regular behavior of DRAM as it was realized in [43]. More specifically, memory bandwidth performance exhibited different rates depending on whether the type of accesses were reads or writes. The cache behavior issue is also further discussed on the next chapter (section 6.2). Additionally, more detailed information regarding experiments on GPU memory caching throughput is provided in chapter B of the appendix.

- Divergent code execution (e.g. kernel *lvmd-krn*) can also cause problems on the prediction process as the execution of such code behaves the same way as the instructions were executed by the all threads belonging to a thread warp. This is another aspect that is not supported by the proposed model as it induces lower performance that predicted.

To the best of the author's knowledge this research work conducts experiments to very wide range of kernels compared to other research in regard to GPU performance modeling. Typically, other related work is limited to a few kernels. The black-box nature of this performance method allows applying it on many kernels without additional effort. In addition, the kernels used in the experiments were selected using justified criteria instead of cherry-picking well suited examples.

All the issues stated above potentially lead to lower performance, excluding the integer instruction cost estimation issue which may lead to higher performance than predicted. However, if a slight change is applied to the prediction method, i.e. using the integer addition instruction cost as a reference for the cost of integer operations, the integer instruction cost could also lead to a higher predicted performance compared to the measured one. Therefore, the proposed method can be additionally used as a reference for guiding optimizations. A kernel designer may choose to use the predicted performance as an indication to the upper bound limit he can achieve by eliminating the bottlenecks. Frequently, the existing bottlenecks are caused by following poor or improper GPU programming practices. In other circumstances, it is the type of the algorithm that does not fit well on GPUs, as it is often the case with irregular parallel algorithms. In either case, the predicted method can prove to be convenient by exposing the theoretically peak attainable performance in case the GPU resources are be optimally utilized.

6. CROSS-VENDOR EVALUATION AND PERFORMANCE LIMITATIONS

In this chapter we explore the potential of applying the proposed performance model on a wider range of hardware, beyond the CUDA capable GPUs. Other GPU hardware vendors for desktop or server computers include AMD and Intel. However, these vendors do not support CUDA. In this regard, in order to be able to apply the same performance prediction method on their hardware requires resolving some portability issues.

6.1 Portability on a different vendor's architecture

So far the performance prediction method has been limited to NVidia GPUs due to employing the CUDA environment. The automation potential of this method relies on the existence of proper profiling metrics along with the capability of micro-benchmarking the particular hardware that is used to assess the device under specific operations including both computation and memory. The latter could easily be applied on a variety of GPU architectures by porting the micro-benchmarks to the OpenCL platform which is supported by multiple vendors. However, the former requirement cannot be easily met as other GPU vendors do not provide the required performance counters so that the automated kernel analysis can be performed and the required kernel parameter values be estimated. In fact, at the time of writing of this thesis, the *nvprof* tool that was employed for CUDA kernel analysis did not support OpenCL kernel profiling. That said, OpenCL kernels cannot be used directly to the purpose of the extraction of kernel parameters. On the other hard, porting applications from CUDA to OpenCL platforms and vice versa is a process that requires significant time and effort. Thus, other possibilities should be explored to this direction.

6.1.1 The HIP/ROCm programming environment

Recently, an alternative programming environment to NVidia CUDA has emerged by AMD named HIP[6, 5], supporting easy migration from CUDA to a platform neutral API. HIP is supported by the ROCm platform which is a full open source software stack and consists of the ROCk kernel driver, the ROCr run-time, the ROCt thunk interface, the HCC compiler and the LLVM-AMDGPU assembler. ROCm platform is based on the HSA specification [82] and it has been extended to support discrete GPUs.

The migration of a CUDA application is assisted by the *HIPify* tool. In this migration the kernel source code itself stays mostly intact, apart from some automated replacements of reserved keywords, which is a key observation as it allows the same kernel source to be examined for performance measurement on other architectures beyond the ones supported by NVidia.

As it is currently the case with OpenCL and HSA environments, the vendor's profiling capabilities provide a set of limited counters preventing the full execution of the performance prediction method on the AMD GPU. For instance, there is no a distinct profiling counter for floating point operations or memory transactions on the AMD GPU. A full list of the supporting profiling counters can be found in [4]. However, as the kernels are essentially the same, the profiling procedure could be performed on the NVidia hardware by using either the CUDA application or the HIP application itself as HIP provides a compatibility layer for both hardware platforms. The same parameters that had been extracted for the previous experiments on the GTX-480 were used for the performance prediction on the AMD GPU.

6.1.2 Kernel parameter portability

This model relies on the principle of parameter portability. The set of defined parameters are generic in nature by hiding special hardware details and thus, the level of the model's abstraction allows retaining hardware dependency. The proposed model is not tailored to a particular hardware though some slight variability on the parameter values between different architectures can be normally expected. However, the overall picture as given by the kernel features is not expected to change. It's the level of abstraction that permits use of the model on different hardware architectures.

NVidia and AMD's GPU architectures are much different, yet they share common features. They are both highly multithreaded and they exploit this multithreading capability in the direction of hiding the expensive device memory access latencies with zero cost context switching. In addition, they both employ flexible SIMD fashion execution so that programming can be applied in a sequential (SISD) form and mapped to SIMD though the hardware or the compiler. Moreover, they both feature fast scratchpad memories, large register sets, L2 caches, coalesced memory accesses, high memory bandwidth, etc. That said, the kernel parameters are not expected to differentiate significantly between vendors' architectures and the feasibility of using these values between architectures will be assessed in the next section.

6.1.3 Experimental results

The HIP programming environment was applied as supported by ROCm 1.4.0 release, on Ubuntu 14.04 Linux 64bit, using an AMD R9-Nano GPU. The theoretical specifications of the particular GPU are provided in table 6.1.

The required micro-benchmark tools were ported to HIP platform with minimal effort through the *HIPify* tool. Thereafter, they were used to generate the R9-Nano GPU parameters so they can be used on the performance model. The produced GPU parameters are shown in table 6.2. The cost weight factors are derived by using these parameter values which are presented in table 6.3.

Table 6.1: Theoretical specifications of the R9-Nano GPU

Feature	Rating	Unit
Memory bandwidth	512	GB/sec
DP Compute	512	GFLOPS
SP Compute	8,190	GFLOPS

Table 6.2: Measured *GPU parameters* for R9-Nano

Parameter	Value
T_{SP} (GFLOPS)	8,032.08
T_{DP} (GFLOPS)	339.84
T_{int} (GIOPS)	1,623.73
T_{add} (GIOPS)	3,985.30
B_{mem} (GB/sec)	430.33
T_{ldst} (GOPS)	1,322.12

The experiments selected were the red/black SOR computation using double precision arithmetic, the single precision GEMM (SGEMM) and the Rodinia *lvmd-krn* kernel. It should noted that the current HIP version is not complete in terms of full coverage of CUDA features as a few features are not supported yet (e.g. use of texture memory). As such, the kernels were selected so that they did not make any use of the unsupported features. The respective parameters for red/black SOR and *lvmd-krn* kernels have already been presented in tables 4.5 and 5.16, respectively. For the SGEMM kernel, the thread block configuration had to be changed to 16×16 as the original implementation employed a 32×32 configuration which is not supported on the AMD platform as the maximum thread block size on the latter was 256 threads per block. Therefore, new parameter data are provided in table 6.4. It is evident that DRAM access traffic has increased due to the smaller blocking.

Running the performance model yields the execution times shown in table 6.5. It is evident that the observed prediction errors were very comparable to the ones produced on NVidia GPUs. Out of the 3 kernels the *lvmd-krn* kernel exhibited slightly higher APE.

In general, it is expected to observe slightly higher APEs on the AMD platform due to the architectural differences between the two different vendor GPU architectures. These differences could slightly differentiate the extracted kernel parameters between the two architectures. However, this is not expected they change dramatically allowing the use of the performance prediction method in cross architecture environments, in the same way it was applied on this experiment. The most notable architectural differences between the two vendors are:

- All NVidia GPUs make use of a logical SIMD width (warp size) equal to 32, whereas AMD GPUs make use of a 64 wide logical SIMD width (wavefront size). This makes AMD GPUs performance more susceptible to thread divergent execution.

Table 6.3: The GPU cost weights as measured for the R9-Nano

Weight	Value
W_{op} (fp32)	1.00
W_{op} (fp64)	23.63
W_{op} (int)	4.95
W_{ldst} (load/store)	3.04
W_{other} (add)	1.01

Table 6.4: SGEMM *kernel parameters* using a 16x16 thread block size.

Parameter	Value
K_{type}	fp32
W_{comp}	1,048,576,000
W_{traf}	61,806,400
E_{mix}	100.00%
D_{ops}	30.19%
D_{ldst}	45.33%
D_{other}	24.48%
O_{krn}	16.97

- It has been observed that AMD ISA is more verbose as it handles all branches explicitly. For instance, execution mask is handled by special instructions (e.g. *s_and_b64 exec,vcc,exec*) and explicit branch instructions (e.g. *s_cbranch_vccz*) are used when conditional statements are executed without divergences between threads in a wavefront. Therefore, when code contains branches it typically generates more control instructions for AMD GPUs to execute.
- Typically, more parallelism is required for full occupancy as each *compute unit* (AMD's counterpart for *Streaming Multiprocessor*) contains 4 SIMD units and each executes on its own threads. This means that an absolute minimum of $64 \times 4 = 256$ active threads are required per *compute unit* in order make it work efficiently, which in turn leads to a minimum of active 16,384 threads for utilizing all compute elements of the R9-Nano GPU.

Having all the architectural differences in mind, the reported error percentages on the experiments are considered to be on par with the previously presented single vendor experiments. The proposed model provides a sufficiently good estimate of the actual execution time.

6.2 Performance model limitations

The proposed prediction technique is able to indirectly capture some characteristics in memory access patterns, as for instance, the degree that memory accesses are being

Table 6.5: Prediction results on the R9-Nano GPU for the red/black SOR, SGEMM and lvmd-krn kernels

Benchmark	Predicted time (msecs)	Measured time (msecs)	Error (%)
red/black SOR	7.75	8.72	-11.18%
SGEMM	0.83	0.94	-11.45%
lvmd-krn	46.27	54.57	-15.21%

coalesced is a measure that is captured. Memory accesses of threads within a warp that map within the same 32 or 128 byte memory segment can coalesce in one or few global memory transactions. Otherwise, when each thread within a warp accesses elements that map to different 32 byte segments then memory accesses are fully uncoalesced and each of them entails a separate 32 byte transaction. Similarly, misaligned accesses can also map to more than one memory transactions. The memory traffic in this work is measured by measuring the amount of transactions as returned by the profiling procedure. Therefore, the actual memory traffic is considered instead of the requested, and thus the effects of uncoalescing are indirectly taken into account. However, other performance related memory access events are not captured. These include shared memory bank conflicts, memory channel conflicts (i.e. partition camping) and constant memory access serialization.

The automated kernel profiling procedure is applied for the total invocations for the particular kernel in the application. This means that the aggregated profiling metrics are considered for the derivation of kernel's parameters. However, many applications tend to be executed for different data size inputs on different executions. In this case the impact of input size on kernel parameters cannot be safely predicted. Either a kernel re-profiling would be required for all possible input sizes or the empirical derivation on the way input size affects the execution time. If the data size is small then the kernel could be under-utilizing the GPU and in this case the proposed model is not able to infer the actual performance. On the other hand, if the data size is sufficiently large then GPU occupancy is considered enough to hide latencies. In this case an extrapolation would be possible by considering the complexity of the computation and having a performance prediction measurement for some reference input data size.

In general, the thread block size configuration stays constant for an application. However, in case it is variable it might affect the kernel parameter values. This fact is induced from the tight connection between threads within a thread block and their data sharing through shared memory. In such cases exploitation of locality is reduced as thread block size gets smaller. This was the case with the SGEMM kernel when using a different thread block size. When a 32×32 thread block was chosen then the features collected led to an operational intensity of 24.81 (table 4.10). In contrast, using a 16×16 thread block configuration increased the amount of memory transactions by +46.26%, leading to a decrement on the operational intensity got 16.97 (table 6.4). However, the different kernel parameters did not significantly affect performance on the particular example due to its compute intensity.

The described performance model assumes the use of simple floating point or integer instructions (additions + multiplications) for performing beneficial operations. These operations can be accurately counted by profiling as performed in the proposed method. If user requires to apply this method on a kernel that employs other special type of beneficial operations e.g. transcendentals (trigonometric functions, square root, etc.), bitwise functions (intense use in hashing algorithms, encryption, etc.), the method can be easily adjusted at the cost of requiring some input by the user. It requires the amount of beneficial operations applied by the kernel and the peak throughput of the target device in this kind of operations. The former could be empirically estimated assuming the user has at least a basic understanding of the computation. The latter could be estimated by either the theoretical device specifications (not always provided for such special operations) or by using a custom benchmark that measures the throughput in the applied kind of operations. If these operations do comprise of multiple instructions the profiler should also be leveraged in order to estimate the type and amount of instructions used for performing each special operation. Thereafter, this information could adjust the respective parameters used in the performance model (D_{ops}, T_{op}, W_{op}) to the particular special operation on the goal of credible performance prediction measurement.

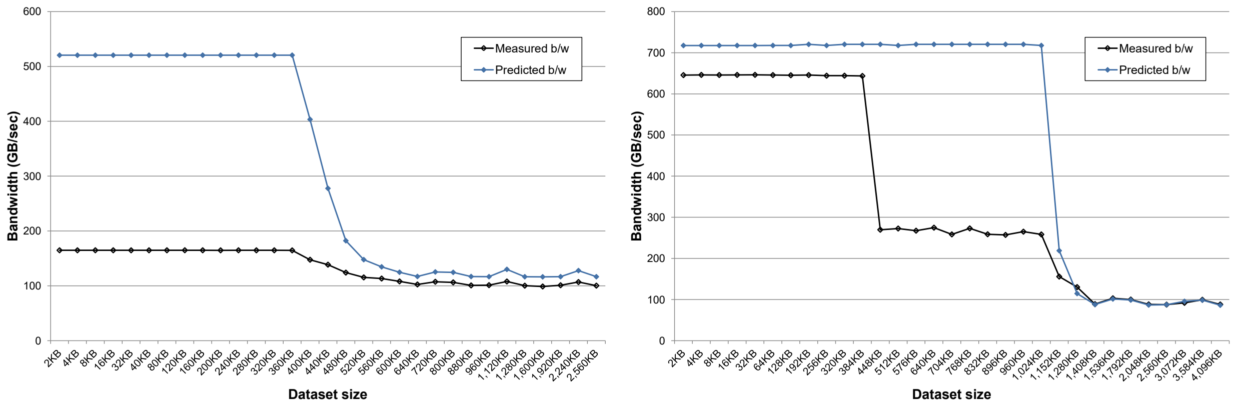
Similarly to the previous issue, the integer operations are significantly more diverse from the perspective of execution cost. This fact can cause misprediction on the results depending on the type of integer operations involved in the GPU kernel code. This issue was earlier discussed in section 5.2.6, as it was encountered on the Rodinia experiments. A typical case is the difference in performance between addition instructions and multiplication instructions, where the performance difference can reach up to 5-fold degree. As the integer instruction profiler counts constitute a summary of all integer instructions, this issue cannot be addressed without providing additional information manually.

As already presented in section 2.2.2, GPUs are typically equipped with L1 and L2 caches, which naturally have limited bandwidth. Thus, if either the L1 or L2 cache utilization poses as a bottleneck, it will not be exposed by the proposed model. However, the benefits of caching on the GPU are not expected to be critical in the same way they are on the CPU and caching is not regularly the bottleneck of the application. This is due to the highly multithreaded nature of GPUs and the typically limited size of GPU caches. In addition, the cache bandwidth behavior is more complex compared to device memory, as different mix of read/write operations can lead to different bandwidth ratios[43]. Using a safe bound for cache memory throughput did not prove as a reliable roofline rule in the experiments. For more detailed information the reader is advised to look in chapter B of the appendix. For this reason, the consideration of caching in the performance model was left out as it would significantly increase the complexity with questionable prediction refinements in return. Therefore, the possibility of exploiting the cached memory transactions in the performance model is left as a future work.

In general, summarizing the proposed performance model provides an optimistic view on the predicted performance of a kernel. The primary limitation is the arbitrary assumption that the kernel execution is not latency bound. Apart from this, there are a few additional limitations worth to note. Memory coalescing is indirectly considered, though a similar behavior between reference and target GPUs is assumed. Data sizes are assumed to stay constant so they don't influence the GPU workloads. The same would apply in cases of random data are used or of variable thread block sizes. Irregular or integer operations could be misinterpreted in regard to its execution cost though there are proposed workarounds. Non similar data caching behavior can also lead to errors in predictions. Other GPU related performance issues are also not considered, the most notable being the shared memory bank conflicts, branch divergences, constant memory serialization. Even further, issues like register bank conflicts of memory channel conflicts are far more complicated to be considered in this high level model. All these issues potentially influence the prediction, though not to a dramatic degree. The most significant features affecting performance are provided by the proposed model and the majority of the mispredictions can be attributed to latency issues.

6.2.1 Exposing limitations through micro-benchmarking

Micro-benchmarks do regularly stress particular resources of the GPU. In this section the performance model is tested against two micro-benchmarks that stress the on-chip memories. These benchmarks [43] are briefly described in B in the appendix. These experiments will be used in order to exhibit some special cases where the model does not evaluate the instruction execution cost properly. It should be noted that all predictions in this section were applied by considering as integer instruction cost (T_{op}) the execution cost of integer addition operations (T_{add}) instead of integer multiply-add (T_{int}) operations. This decision is based on the fact that the use of multiply-add instructions is minimal in this applications.



(a) GTX-660 **(b) GTX-960**

Figure 6.1: Cachebench bandwidth predictions and measurements on various data-sets.

In figure 6.1 the benchmark assessing memory bandwidth (vertical axis) of global memory hierarchy (*cachebench*) for repetitively reading 32bit data elements from a data array with various sizes (horizontal axis). This benchmark explores the bandwidth limitations of L1, L2 caches and GPU DRAM. The measured bandwidth is shown along with the predicted bandwidth, which was estimated by using the predicted execution time given through the proposed model. Benchmarks were executed on the GTX-660 and GTX-960 GPUs.

In both figures it is apparent that when the data-set is large so that cache memories cannot fit the data, the prediction runs accurately. However, for small data-sets ($\leq 360KB$) the prediction is particularly bad on the GTX-660. This is justified by the fact that for the particular cases the performance model accounts just the instruction execution cost, as no DRAM transactions are observed and all transactions are served by the L2 cache, which is officially sized to 384KB on this GPU. For the GTX-960 the prediction error is significantly less for small data-sets, as the L1/texture caches serve the majority of data accesses. However, for medium sized data-sets the prediction is equally bad as with the previous GPU, as the L1 caches are too small to accommodate the data-set and the data accesses are served by the L2 cache. That said, the intensive cache usage on GPU applications may induce errors with the proposed model.

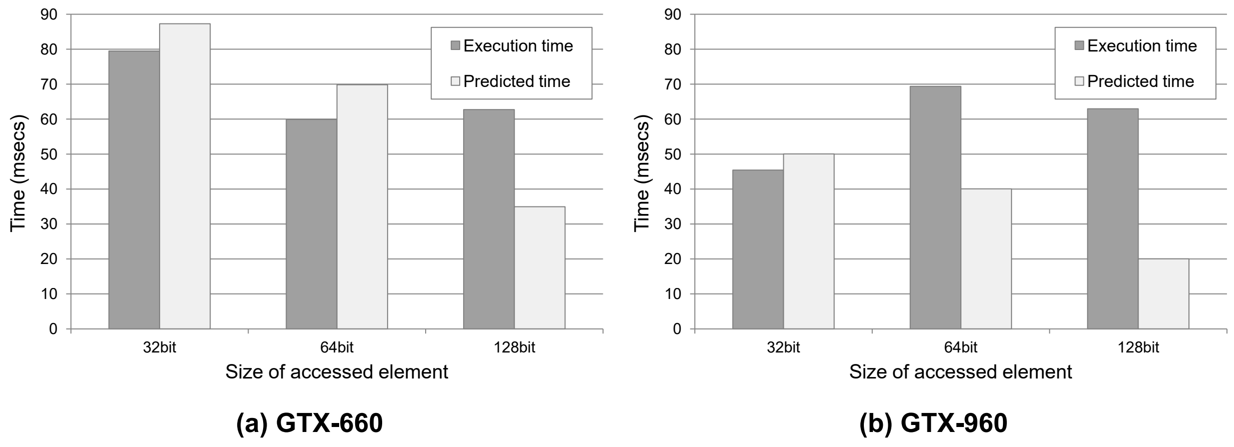


Figure 6.2: Shmembench predictions and measurements for various data type sizes.

In figure 6.2 the shared memory benchmark (*shmembench*) was applied on the same GPUs. This benchmark measures the effective bandwidth on GPU by intensively swapping memory elements in shared memory, avoiding bank conflict effects. The source code of this benchmark is provided in section A.4 of the appendix. The benchmark is executed for 32bit, 64bit and 128bit elements, which translate to shared memory loads of the respective size. In all three cases the same amount of data is transferred to/from shared memory. The GTX660 seems to exhibit performance close to prediction for 32bit and 64bit accesses. In fact, measured performance was better than the predicted performance due to the multi-issue capability of load/store and integer instructions. This happens because the CC3.0 SMs can serve 32 element accesses per clock on shared memory, with up to 64bit element size. Prediction is significantly worse on the last case as the cost of loading 128bit elements is the double of the cost of loading 64bit elements. The GTX960 differs on the specifications on that its SMs can serve 32 element accesses per clock, with up to 32bit element size. That said, the observed prediction error is expected in both 64bit and 128bit cases, as shown in the experiments.

7. CONCLUSIONS AND FUTURE WORK

In the concluding chapter an overview of the key highlights of the proposed model is provided along with a discussion on the possible future directions for improvement.

7.1 Conclusions

During last years the GPUs have found a solid spot in the high performance computing industry. Performance of compute intensive parts of code can be frequently accelerated to a dramatic degree. The nature of GPUs, however, enables applications to exhibit highly variable performance as it can be significantly affected by many factors. In addition GPUs tend to be based on various GPU architectures and not being bound on a particular one like CPUs do with the prevalence of x86-64 architecture. The exploitation of GPUs as compute accelerators raises the significance of performance prediction methods on these devices.

This thesis presents an analytical performance model that derives from the roofline model [92]. The roofline principle was validated on GPUs by developing a micro-benchmark through which performance was investigated on a wide range of operation intensity values. Through a quantitative approach, the proposed model is able to provide timings that approximate actual execution measurements on real hardware. In addition, an alternative visual representation approach was presented, named *quadrant-split*, which is insightful in cases of multiple compute devices being represented along with a single application characterized by a particular operation intensity. The merit of the model's simplicity and its high abstraction characteristic allows providing results, final and intermediate, that can be easily interpreted by the final developer by being more human friendly. The small amount of required parameters pose the method as readily applicable.

One of the key points of the proposed method is the ability to extract the kernel's parameters by exploiting a mere set of profiling metrics as input parameters. This is captured through a kernel profiling procedure in a *black box* fashion. Any internal knowledge of the kernel structure itself is not required by the developer. Furthermore, the proposed method can be developed as an automated tool which is executed without intervention from the developer. In this regard, the developer can apply the method on kernels and use it as a guidance tool without previous inspection the kernel design itself.

The proposed method achieves a better understanding of both compute and memory workloads compared to a pure theoretical peak approach primarily for two reasons. First, both the execution of non-essential and load/store instructions are considered by modeling their implications in the instruction pipeline and thus, their impact on the effective peak performance on beneficial instructions. Additionally, the type of mix of compute operations is also taken into account, i.e. the proportion of effective multiply-add operations in total amount of compute instructions. Second, the memory traffic requirements are measured by considering the actual traffic which means that any trivial locality is being indirectly accounted as cached transactions will not produce any DRAM traffic and the degree of memory access coalescing is also accounted as it directly affects the conducted memory traffic. The model provides an adjusted roofline on the peak performance based on these considerations.

The proposed performance model was tested and validated on a wide range of real world kernels. It was applied on stencil computations (red/black SOR and LMSOR), matrix multiplication and a wide range of Rodinia benchmark kernels. The model was also applied on the *mixbench* micro-benchmark and it exhibited very accurate results. Furthermore, it was also tested for cross-vendor applicability on the HIP programming environment [6, 5] of the ROCm platform which is developed by AMD. The results were quite promising as they were similar to CUDA prediction in terms of absolute errors, despite the broader architectural differences between different vendor GPUs. The exact performance is dependent on special issues as the exact instruction mix, variability in cache behavior, pipeline latencies, the available parallelism and additional latencies which push performance to lower levels than the predicted ones, as the proposed model does not take into account these factors. Nevertheless, in these cases the performance prediction measurements serve as an upper bound performance and they indicate the potential room for improvement with further optimizations.

7.2 Future work discussion and proposed model refinements

In this section some directions are provided towards the improvement of the proposed performance model. A primary obvious improvement would focus on optimizing the prediction accuracy of the model. Some possible improvements could move around the observed limitations of the model that have already been discussed. For instance, these include the more accurate consideration of instruction execution cost, the quantification of thread divergent execution implications and potential reduced parallelism due to other serialization factors, e.g. shared memory conflicts, constant memory access serializations. In addition, the short kernel execution amplifies the effect of kernel invocation overhead which should be included in the model, as well. The variability of cache effects should also be considered as it potentially differentiates the amount of conducted DRAM accesses between various types of GPUs.

However, before proposing any possible directions for improvement, there are a few key questions that arise:

- What is the current hardware support of additional kernel profiling input that could potentially be exploited by the performance model?
- Could the user provide additional input in order to fine tune the prediction results and which one would be candidate to this purpose?
- How stable would an additional kernel parameter be across multiple hardware architectures?

- To what degree the additional complexity on the performance model refinement would affect its level of abstraction?

Extending the model to a particular direction should be made by having considered the questions above. Nevertheless, in this chapter some ideas on overcoming the model's limitations will be discussed. These proposals may require additional input either from the profiler or the developer (manual input of data). Finally, a potential improvement of the method will be discussed from the perspective of implementation.

7.2.1 Additional input parameters

The NVidia profiling capabilities provide a rich set of metrics that cover most aspects of kernel execution. In this proposed model a restricted set of parameters have been employed that make use of a small subset of the available metrics (table 4.1). Special cases could potentially be addressed by the exploitation of additional metrics.

Nevertheless, extending the model with more parameters would increase its complexity and a proportional number of additional benchmarks would be required for target GPUs. Thus, this decision is a trade-off between simplicity and complexity. The current form of the model features the advantage of simplicity, so it is proposed to apply refinements selectively and progressively. This strategy allows focusing on specific issues only when required. Of course, this process requires the understanding of kernel's inner performance characteristics. The author's suggestion is to extend the model only on special cases.

Shared memory bank conflicts or wide shared memory accesses (>32bit) are an example that could be potentially better predicted by using additional metrics. More specifically, the NVidia profiling capabilities provide a set of shared memory profiling counter metrics (*shared_load_transactions*, *shared_store_transactions*, *shared_efficiency*) that could represent the actual shared memory traffic and efficiency. The use of these counters could be employed in order to map such shared memory effects as effectively increasing the number of issued load/store instructions. The performance model could be easily modified in order to reflect this increment. All recent NVidia architectures seem to share common shared memory characteristics and the behavior of shared memory is expected to be similar between various GPUs.

On other cases the available profiling metrics seem not to be adequate in order to provide additional model parameters. A particular example is the constant memory access serialization issue which is not considered by the model. In this regard no profiling metrics were found in order to incorporate constant memory serialization effects, which may occur when multiple threads of a warp access constant memory using different indices. A profiling counter is proposed, exposing the constant memory access efficiency (e.g. *constant_efficiency*). This could be exploited by the performance model in order to take these effects under consideration.

Another case could be the use of special, non-distinctively profiled compute operations. The typical use of the performance model involves compute operations which can be directly profiled. Regular floating point operations (additions and multiplications) of single or double precision counts are quite accurately measured with the provided counters (i.e. *flop_count_XXX*) on NVidia GPUs. The cost of execution of such operations is also smoothly measured via micro-benchmarking. On specific special occasions, however, as it has also been discussed in section 6.2, the operation counts cannot be such promptly measured. Moreover, the assessment of their execution cost requires special tailored micro-benchmarking. Some examples are division and transcendental operations, non-typical integer operations (e.g. bitwise manipulations, divisions, etc.). The same issue holds for integer operations in general, which may significantly vary in terms of cost depending on their type of operations, i.e. being additions, multiplications or bitwise operations. In order to provide an automated prediction method, both a micro-benchmark tool for the particular type of operations and an appropriate profiling counter metric measuring these operations will be required. Therefore, some individual counter metrics that could be proposed to be supported by future hardware and software would be metrics specializing on integer addition, multiplication, bitwise operations, as well as, sinusoidals, division and square root.

In cases where the available metrics are insufficient for assessing a parameter, the human factor is left as an alternative source for estimating valid values of individual parameters. Nonetheless, this alternative requires good experience and knowledge of the kernel's inner workings, a requirement that decimates one of the advantages of this method.

A particular example is cache behavior, which cannot be predicted in a straightforward fashion. Its implications have already been discussed on the previous chapter. What could be suggested is letting the developer determining the expected amount of DRAM traffic to be conducted by the kernel (W_{traf} parameter). Beyond the kernel's memory access patterns, this strategy requires knowledge of the GPU cache memory characteristics, as well as, memory access coalescing implications.

The same workaround can also be applied in cases where the memory transaction granularity is different between the reference and the target GPUs. Global memory transactions are accounted by profiling the total amount of read and write transactions on the reference GPU. This assumes that transactions are of the same width on both the reference and the target GPUs. However, this is not always true, e.g. on Fermi GPUs they can be applied as 128 byte multiples (4x32) when cached on the L1 or as single 32 byte transactions otherwise. This fact potentially causes discrepancy on the actual amount of transactions seen on the reference and the target GPUs, which in turn might negatively affect the prediction accuracy. A user provided amount of DRAM traffic could be a possible workaround.

7.2.2 Simulated parameter extraction

Improving the method's applicability is possible through the use of simulated execution of GPU programs. The ultimate goal is to decouple the requirements of specific hardware counters from the kernel parameters employed by the method. The current form of the method relies on the availability of rich profiling capabilities of GPU hardware and software.

To overcome the requirements of using a real GPU device that offers a capable set performance counters, a software approach for the parameter extraction could be employed. Such move would avoid the requirement of having to use a real GPU device for conducting the profiling process. This software could be a GPU simulator or a GPU code interpreter, capable of executing compiled GPU programs and extracting the requested execution information by software analysis. For instance, two candidate interpreters and simulators that have already been presented in the related work (section 2.4.1) are GPGPU-Sim [9] and Oclgrind [81]. A major advantage of this method is the flexibility of extending the performance counters with capabilities that are not offered by the current metrics on real hardware (e.g. distinct counters for integer multiplications and other integer operations). On the other hand, of course, the estimation of parameters is expected to be a significantly slower process with this approach.

ABBREVIATIONS - ACRONYMS

ALU	Arithmetic Logic Unit
API	Application Programming Interface
ASIC	Application-specific integrated circuit
CPU	Central Processing Unit
CTA	Cooperative Thread Array
DLP	Data Level Parallelism
FLOPS	Floating Point Operations
GPU	Graphics Processing Unit
HPC	High Performance Computing
ILP	Instruction Level Parallelism
ISA	Instruction Set Architecture
JIT	Just In Time compilation
LMSOR	Local Modified SOR
MIMD	Multiple-Instruction, Multiple-Data
MLP	Memory Level Parallelism
PDE	Partial Differential Equation
SIMD	Single-Instruction, Multiple-Data
SIMT	Single-Instruction, Multiple-Thread
SM	Streaming Multiprocessor
SOR	Successive Over-Relaxation
SP	Streaming Processor
SPIR	Standard Portable Intermediate Representation
TLP	Thread Level Parallelism

APPENDIX A. MICRO-BENCHMARK KERNEL SOURCE CODES

A.1 Roofline approximation (mixbench)

```
template <class T, int blockSize, unsigned int granularity,
          unsigned int fusion_degree, unsigned int compute_iterations
          unsigned int compute_iterations>
__global__ void benchmark_func(T seed, T *g_data){
        const int stride = blockSize;
        int idx = blockIdx.x*blockSize*granularity + threadIdx.x;
        const int big_stride = gridDim.x*blockSize*granularity;

        T tmps[granularity];
        for(int k=0; k<fusion_degree; k++){
                #pragma unroll
                for(int j=0; j<granularity; j++){
                        // Load elements (memory intensive part)
                        tmps[j] = g_data[idx+j*stride+k*big_stride];
                        // Perform computations (compute intensive part)
                        for(int i=0; i<compute_iterations; i++){
                                tmps[j] = tmps[j]*tmps[j]+seed;
                        }
                }
                // Multiply add reduction
                T sum = (T)0;
                #pragma unroll
                for(int j=0; j<granularity; j+=2)
                        sum += tmps[j]*tmps[j+1];
                // Dummy code
                if( sum==(T)-1 ) // Designed so it never executes
                        g_data[idx+k*big_stride] = sum;
        }
}
```

A.2 Compute throughput evaluation

```
template <class T, class F>
__global__ void benchmark_func(T *g_data){
        F func;
        int tid = threadIdx.x;
        T r0 = g_data[blockIdx.x*blockDim.x + tid],
          r1 = r0+(T)(31),
          r2 = r0+(T)(37),
          r3 = r0+(T)(41),
          r4 = r0+(T)(43),
          r5 = r0+(T)(47),
          r6 = r0+(T)(53),
          r7 = r0+(T)(59),
          r8 = r0+(T)(61),
          r9 = r0+(T)(67),
          rA = r0+(T)(71),
          rB = r0+(T)(73),
          rC = r0+(T)(79),
          rD = r0+(T)(83),
          rE = r0+(T)(89),
          rF = r0+(T)(97);
        #pragma unroll 32
        for(int j=0; j<COMP_ITERATIONS; j++){
                r0 = func(r6, r6, r7);
                r1 = func(r7, r7, r8);
                r2 = func(r8, r8, r9);
                r3 = func(r9, r9, rA);
                r4 = func(rA, rA, rB);
                r5 = func(rB, rB, rC);
                r6 = func(rC, rC, rD);
                r7 = func(rD, rD, rE);
                r8 = func(rE, rE, rF);
                r9 = func(rF, rF, r0);
                rA = func(r0, r0, r1);
                rB = func(r1, r1, r2);
                rC = func(r2, r2, r3);
                rD = func(r3, r3, r4);
                rE = func(r4, r4, r5);
                rF = func(r5, r5, r6);
        }
        if(r0==(T)-123456789.123456789){ // extremely unlikely to happen
                g_data[tid+0*warpSize] = r0+r1+r2+r3+r4+r5+r6+r7+
                                         r8+r9+rA+rB+rC+rD+rE+rF;
        }
}
```

T is a template parameter designating the data type involved in the assessed operation and *func* is a functor that designates the particular type of operation, e.g. for assessing single precision multiply-add operations *float* should be passed as parameter *T* and the *dev_fun_mad* class is passed as parameter *F*:

```
template<class T>
class dev_fun_mad{
public:
        __device__ T operator()(T v1, T v2, T v3){ return v1*v2+v3; }
};
```

A.3 Memory bandwidth evaluation

```
template <class T, int granularity, bool doRead, bool doWrite, bool useTexture>
__global__ void kmemaccess(const T value, const T * __restrict g_adata,
                           T * __restrict g_cdata){
        const unsigned int blockSize = blockDim.x;
        const int stride = blockSize;
        int i = blockIdx.x*blockSize*granularity + threadIdx.x;
        T tmps[granularity];
        #pragma unroll
        for(int j=0; j<granularity; j++)
```

```
                tmps[j] = doRead ?
                          (useTexture ? textureFetch<T>(i+j*stride) :
                                        g_adata[i+j*stride]) : value;
        T sum = tmps[0];
        #pragma unroll
        for(int j=1; j<granularity; j++)
                sum = sum + tmps[j];
        if( doWrite || value==sum ){ // always: value!=sum
                #pragma unroll
                for(int j=0; j<granularity; j++)
                        g_cdata[i+j*stride] = tmps[j];
        }
}
```

Template parameter *T* designates the data type used for memory access and the *granularity* parameter was set to 8 in the experiments. The parameters *doRead*, *doWrite* and *useTexture* are used to determine whether the benchmark shall involve memory loads, memory stores or/and texture fetches, respectively.

The example below shows how the template function *textureFetch* is defined for *double* data types, for which a native texture fetch function is not provided in CUDA:

```
template <class T>
__device__ __forceinline__ T textureFetch(unsigned int Aindex){
}

template <>
__device__ __forceinline__
double textureFetch<double>(unsigned int Aindex){
        int2 v = tex1Dfetch(texdataD, Aindex);
        return __hiloint2double(v.y, v.x);
}
```

A.4 Load/store operation throughput evaluation

```
template <class T>
__global__ void benchmark_shmem(T *g_data){
        T *shm_buffer = (T*)shm_buffer_ptr;
        int tid = threadIdx.x;
        int globaltid = blockIdx.x*blockDim.x + tid;
        set_vector(shm_buffer, tid+0*blockDim.x, init_val<T>(tid));
        set_vector(shm_buffer, tid+1*blockDim.x, init_val<T>(tid+1));
        set_vector(shm_buffer, tid+2*blockDim.x, init_val<T>(tid+3));
        set_vector(shm_buffer, tid+3*blockDim.x, init_val<T>(tid+7));
        set_vector(shm_buffer, tid+4*blockDim.x, init_val<T>(tid+13));
        set_vector(shm_buffer, tid+5*blockDim.x, init_val<T>(tid+17));
        __threadfence_block();
        #pragma unroll 32
        for(int j=0; j<TOTAL_ITERATIONS; j++){
                shmem_swap(shm_buffer+tid+0*blockDim.x,
                           shm_buffer+tid+1*blockDim.x);
                shmem_swap(shm_buffer+tid+2*blockDim.x,
                           shm_buffer+tid+3*blockDim.x);
                shmem_swap(shm_buffer+tid+4*blockDim.x,
                           shm_buffer+tid+5*blockDim.x);
                __threadfence_block();
                shmem_swap(shm_buffer+tid+1*blockDim.x,
                           shm_buffer+tid+2*blockDim.x);
                shmem_swap(shm_buffer+tid+3*blockDim.x,
                           shm_buffer+tid+4*blockDim.x);
                __threadfence_block();
        }
        g_data[globaltid] = reduce_vector<T>(shm_buffer[tid+0*blockDim.x],
                                             shm_buffer[tid+1*blockDim.x],
                                             shm_buffer[tid+2*blockDim.x],
                                             shm_buffer[tid+3*blockDim.x],
                                             shm_buffer[tid+4*blockDim.x],
                                             shm_buffer[tid+5*blockDim.x]);
}
```

The templated functions *init_val*, *set_vector*, *shmem_swap* and *reduce_vector* are used to allow the kernel to be compiled for multiple data types. For instance, the specialized functions for data type *float2* are shown below:

```
template <>
__device__ float2 init_val(int i){
        return make_float2(i, i+11);
}

template <>
__device__ void set_vector(float2 *target, int offset, float2 v){
        target[offset].x = v.x;
        target[offset].y = v.y;
}

template <class T>
__device__ void shmem_swap(T *v1, T *v2){
        T tmp;
        tmp = *v2;
        *v2 = *v1;
        *v1 = tmp;
}

template <>
__device__ float2 reduce_vector(float2 v1, float2 v2, float2 v3,
                                float2 v4, float2 v5, float2 v6){
        return make_float2(v1.x + v2.x + v3.x + v4.x + v5.x + v6.x,
                           v1.y + v2.y + v3.y + v4.y + v5.y + v6.y);
}
```

APPENDIX B. BENCHMARKING OF FAST ON-CHIP GPU MEMORIES

A diagram with the typical GPU cache memory hierarchy is depicted on figure B.1. This hierarchy is either explicitly or implicitly visible to the programmer through the global, texture, constant and shared memories that CUDA provides. GPU memory type characteristics are summarized in table B.1. Other vendors provide other special memory types as well, e.g. last generations of AMD GPUs include a fast scratchpad shared between all compute units on the GPU (Global Data Share, GDS[2]).

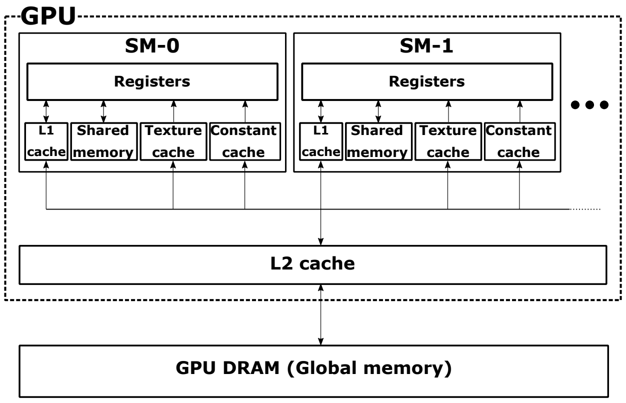


Figure B.1: GPU memory hierarchy includes caches and scratchpads.

In [43] the on-chip memories of the GPU were assessed by micro-benchmarking. These micro-benchmarks were designed with a focus on stressing the particular special memory resources of the GPU. They consist of *cachebench*, *shmembench* and *constbench*, for the assessment of L1-L2 & texture cache bandwidth, shared memory bandwidth and constant cache memory bandwidth, respectively. All micro-benchmark source codes are freely available for experimentation[1] under the GNU GPL 2 license.

Table B.1: The GPU on-chip memory types as provided by modern CUDA GPUs

	GPU physical memory storage		
Memory type	Part of the SM	Scratchpad	Read-only
Shared memory	✓	✓	
L1 cache	✓		✓*
L2 cache			
Texture cache	✓		✓
Constant cache	✓		✓

*GPUs currently do not cache global memory stores in the L1 cache though values of spilled registers and local arrays can be cached.

A brief description of each micro-benchmark application follows.

[1]http://github.com/ekondis/gpumembench/releases/tag/v0.1

B.1 cachebench (L1, L2 & texture cache micro-benchmark)

Benchmarking the multilevel cache hierarchy is achieved by a kernel in which threads of a warp access elements repeatedly in a sequence using strides equal to thread block size (figure B.2). Strides are applied for a predefined number of iterations (threshold) which is set at the compilation time and it virtually determines the size of the data-set that is being accessed. Afterwards, the index is reset to its initial position and the procedure continues repeatedly. For data-sets smaller than the size of grid of threads some of the indexes of different threads are overlapped. Each thread performs a large total number of accesses (8192) in order to eliminate the relative extra overhead of other instructions and an adequate amount of thread blocks is created at the invocation in order to keep the GPU device occupied. A few experiments are conducted in which the threshold is progressively increased. The larger the threshold is, the bigger the data-set that is intensively accessed.

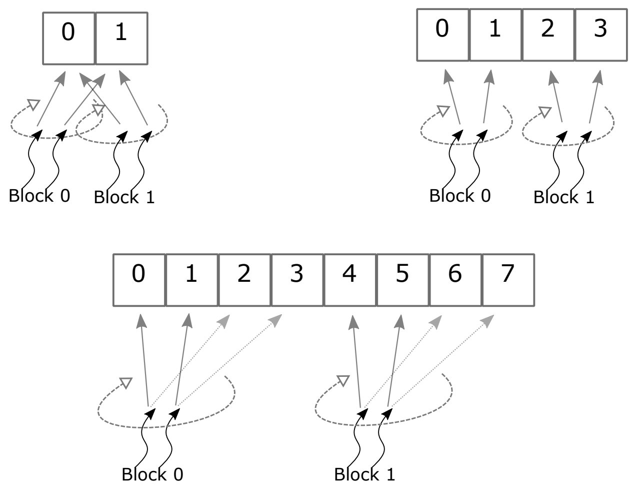


Figure B.2: Thread accesses in cachebench under a simplified scenario with 3 different configurations.

The kernel is implemented by either reading elements directly from global memory or by reading elements via texture memory. Thus, all types of generic caches can be assessed, i.e. L1 cache, L2 cache and texture cache.

B.2 shmembench (shared memory micro-benchmark)

The shared memory micro-benchmark works by exchanging repeatedly scalar/vector values in shared memory. Each swap amounts to two load and two store accesses. Thread synchronization barriers are also limited to *__threadfence_block()* as the primary goal in the kernel is to evaluate the throughput of shared memory accesses through data exchanges and not to involve extra synchronization overheads. Element accesses are sequential between threads in a warp thus no bank conflicts occur. Each thread performs a total of $5 \times 1024 = 5K$ element swaps in shared memory.

B.3 constbench (constant cache micro-benchmark)

The estimation of constant memory bandwidth relies on a large number of threads, all of which perform the exact the same series of memory loads in a sequential pattern, in a constant memory region. The constant memory data-set consists of 1024 elements and every thread reads all elements and calculates their sum. As all threads in a warp access the same elements per time-step the constant memory cache broadcasts the element value to all threads of the thread warp which is the intended use of constant memory and no serialization occurs. The calculation is performed in multiple iterations and all threads perform the same computation.

BIBLIOGRAPHY

[1] Y. Abe, H. Sasaki, S. Kato, K. Inoue, M. Edahiro, and M. Peres. Power and performance characterization and modeling of gpu-accelerated systems. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 113–122, May 2014.

[2] Inc. Advanced Micro Devices. AMD Graphics Cores Next (GCN) Architecture. `http://www.amd.com/Documents/GCN_Architecture_whitepaper.pdf`, Jun 2012.

[3] AMD. *Reference Guide: AMD Intermediate Language (IL)*, Oct 2011. Rev. 2.4.

[4] AMD. *User Guide, AMD GPU Performance API*, 2015.

[5] AMD. HIP. `https://github.com/GPUOpen-ProfessionalCompute-Tools/HIP`, 2016.

[6] AMD. *HIP Data Sheet*, 2016. Rev. 1.7.

[7] AMD. ROCm - Open Source Platform for HPC and Ultrascale GPU Computing. `https://github.com/RadeonOpenCompute/ROCm/tree/roc-1.4.0`, 2016.

[8] Sara S. Baghsorkhi, Matthieu Delahaye, Sanjay J. Patel, William D. Gropp, and Wen-mei W. Hwu. An adaptive performance modeling tool for gpu architectures. *SIGPLAN Not.*, 45(5):105–114, January 2010.

[9] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt. Analyzing cuda workloads using a detailed gpu simulator. In *2009 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 163–174, April 2009.

[10] P. F. Baumeister, T. Hater, J. Kraus, D. Pleiter, and P. Wahl. A performance model for gpu-accelerated fdtd applications. In *2015 IEEE 22nd International Conference on High Performance Computing (HiPC)*, pages 185–193, Dec 2015.

[11] Shajulin Benedict, R. S. Rejitha, and Suja A. Alex. Energy and performance prediction of cuda applications using dynamic regression models. In *Proceedings of the 9th India Software Engineering Conference*, ISEC '16, pages 37–47, New York, NY, USA, 2016. ACM.

[12] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, August 2011.

[13] N. Bombieri, F. Busato, and F. Fummi. A fine-grained performance model for gpu architectures. In *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1267–1272, March 2016.

[14] L. A. Boukas and N. M. Missirlis. The parallel local modified sor for nonsymmetric linear systems. *International Journal of Computer Mathematics*, 68(1-2):153–174, 1998.

[15] M. Boyer, J. Meng, and K. Kumaran. Improving gpu performance prediction with data transfer modeling. In *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2013 IEEE 27th International*, pages 1097–1106, May 2013.

[16] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for gpus: Stream computing on graphics hardware. *ACM Trans. Graph.*, 23(3):777–786, August 2004.

[17] Guillaume Chapuis, Stephan Eidenbenz, and Nandakishore Santhi. Gpu performance prediction through parallel discrete event simulation and common sense. In *Proceedings of the 9th EAI International Conference on Performance Evaluation Methodologies and Tools*, VALUETOOLS'15, pages 204–211, ICST, Brussels, Belgium, Belgium, 2016. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[18] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 44–54, Oct 2009.

[19] J. Choi, M. Dukhan, X. Liu, and R. Vuduc. Algorithmic time, energy, and power on candidate hpc compute building blocks. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 447–457, May 2014.

[20] Yiannis Cotronis, Elias Konstantinidis, Maria A. Louka, and Nikolaos M. Missirlis. *Parallel SOR for Solving the Convection Diffusion Equation Using GPUs with CUDA*, pages 575–586. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[21] Yiannis Cotronis, Elias Konstantinidis, Maria A. Louka, and Nikolaos M. Missirlis. A comparison of CPU and GPU implementations for solving the convection diffusion equation using the local modified SOR method. *Parallel Computing*, 40(7):173 – 185, 2014. 7th Workshop on Parallel Matrix Algorithms and Applications.

[22] Yiannis Cotronis, Elias Konstantinidis, and Nikolaos M. Missirlis. *A GPU Implementation for Solving the Convection Diffusion Equation Using the Local Modified SOR Method*, pages 207–221. Springer International Publishing, Cham, 2014.

[23] Leonardo Dagum and Ramesh Menon. Openmp: an industry standard api for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.

[24] T. T. Dao, J. Kim, S. Seo, B. Egger, and J. Lee. A performance model for gpus with caches. *IEEE Transactions on Parallel and Distributed Systems*, 26(7):1800–1813, July 2015.

[25] Tor Dokken, Trond R. Hagen, and Jon M. Hjelmervik. The gpu as a high performance computational resource. In *Proceedings of the 21st Spring Conference on Computer Graphics*, SCCG '05, pages 21–26, New York, NY, USA, 2005. ACM.

[26] Christian Feichtinger, Johannes Habich, Harald Köstler, Ulrich Rüde, and Takayuki Aoki. Performance modeling and analysis of heterogeneous lattice boltzmann simulations on cpu–gpu clusters. *Parallel Computing*, 46:1 – 13, 2015.

[27] Steven Fortune and James Wyllie. Parallelism in random access machines. In *Proceedings of the Tenth Annual ACM Symposium on Theory of Computing*, STOC '78, pages 114–118, New York, NY, USA, 1978. ACM.

[28] Message P Forum. Mpi: A message-passing interface standard. Technical report, Knoxville, TN, USA, 1994.

[29] P. Guo, L. Wang, and P. Chen. A performance modeling and optimization analysis tool for sparse matrix-vector multiplication on gpus. *IEEE Transactions on Parallel and Distributed Systems*, 25(5):1112–1123, May 2014.

[30] Torsten Hoefler, William Gropp, William Kramer, and Marc Snir. Performance modeling for systematic performance tuning. In *State of the Practice Reports*, SC '11, pages 6:1–6:12, New York, NY, USA, 2011. ACM.

[31] Sunpyo Hong and Hyesoon Kim. An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 152–163, New York, NY, USA, 2009. ACM.

[32] Sunpyo Hong and Hyesoon Kim. An integrated gpu power and performance model. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 280–289, New York, NY, USA, 2010. ACM.

[33] HSA foundation. *HSA Programmer's Reference Manual: HSAIL Virtual ISA and Programming Model, Compiler Writer, and Object Format (BRIG), Version 1.1*, Feb 2016.

[34] N. Jacob and C. Brodley. Offloading ids computation to the gpu. In *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)*, pages 371–380, Dec 2006.

[35] Google John Kessenich and Intel Boaz Ouriel. *SPIR-V Specification, Version 1.00, Revision 9*. Khronos group, Feb 2017.

[36] Ali Karami, Farshad Khunjush, and Seyyed Ali Mirsoleimani. A statistical performance analyzer framework for opencl kernels on nvidia gpus. *The Journal of Supercomputing*, 71(8):2900–2921, 2015.

[37] A. Kerr, G. Diamos, and S. Yalamanchili. A characterization and analysis of ptx kernels. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pages 3–12, Oct 2009.

[38] John Kessenich, Dave Baldwin, and Randi Rost. The opengl shading language. *Language version*, 1, 2004.

[39] Khronos group. *The OpenCL Specification*, 2009.

[40] Y. Kim and A. Shrivastava. Cumapz: A tool to analyze memory access patterns in cuda. In *2011 48th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 128–133, June 2011.

[41] Poonacha Kongetira, Kathirgamar Aingaran, and Kunle Olukotun. Niagara: A 32-way multithreaded sparc processor. *IEEE Micro*, 25(2):21–29, March 2005.

[42] E. Konstantinidis and Y. Cotronis. A practical performance model for compute and memory bound GPU kernels. In *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 651–658, March 2015.

[43] E. Konstantinidis and Y. Cotronis. A quantitative performance evaluation of fast on-chip memories of GPUs. In *2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, pages 448–455, Feb 2016.

[44] Elias Konstantinidis and Yiannis Cotronis. Accelerating the red/black sor method using gpus with cuda. In Roman Wyrzykowski, Jack Dongarra, Konrad Karczewski, and Jerzy Waśniewski, editors, *Parallel Processing and Applied Mathematics: 9th International Conference, PPAM 2011, Torun, Poland, September 11-14, 2011. Revised Selected Papers, Part I*, pages 589–598, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[45] Elias Konstantinidis and Yiannis Cotronis. Graphics processing unit acceleration of the red/black sor method. *Concurrency and Computation: Practice and Experience*, 25(8):1107–1120, 2013.

[46] Elias Konstantinidis and Yiannis Cotronis. A quantitative roofline model for GPU kernel performance estimation using micro-benchmarks and hardware metric profiling. *Journal of Parallel and Distributed Computing*, 107:37 – 56, 2017.

[47] K. Kothapalli, R. Mukherjee, M. S. Rehman, S. Patidar, P. J. Narayanan, and K. Srinathan. A performance prediction model for the cuda gpgpu platform. In *2009 International Conference on High Performance Computing (HiPC)*, pages 463–472, Dec 2009.

[48] R.D. Chamberlain L. Ma, K. Agrawal. A memory access model for highly-threaded many-core architectures. *Future Generation Computer Systems*, 30:202–215, 2014.

[49] Chris Lattner and Vikram Adve. The llvm compiler framework and infrastructure tutorial. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 15–16. Springer, 2004.

[50] Aaron Lefohn, Ian Buck, John D Owens, and Robert Strzodka. Gpgpu: General-purpose computation on graphics processors. *Presentation at IEEE Visualization*, 2004.

[51] J. Lemeire, J. G. Cornelis, and L. Segers. Microbenchmarks for gpu characteristics: The occupancy roofline and the pipeline model. In *2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, pages 456–463, Feb 2016.

[52] A. Li, S. L. Song, E. Brugel, A. Kumar, D. Chavarría-Miranda, and H. Corporaal. X: A comprehensive analytic model for parallel machines. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 242–252, May 2016.

[53] K. Li, W. Yang, and K. Li. Performance analysis and optimization for spmv on gpu using probabilistic modeling. *IEEE Transactions on Parallel and Distributed Systems*, 26(1):196–205, Jan 2015.

[54] Teng Li, Vikram K. Narayana, and Tarek El-Ghazawi. Symbiotic scheduling of concurrent gpu kernels for performance and energy optimizations. In *Proceedings of the 11th ACM Conference on Computing Frontiers*, CF '14, pages 36:1–36:10, New York, NY, USA, 2014. ACM.

[55] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. Nvidia tesla: A unified graphics and computing architecture. *IEEE Micro*, 28(2):39–55, March 2008.

[56] Erik Lindholm, Mark J. Kilgard, and Henry Moreton. A user-programmable vertex engine. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '01, pages 149–158, New York, NY, USA, 2001. ACM.

[57] W. Liu, W. Muller-Wittig, and B. Schmidt. Performance predictions for general-purpose computation on gpus. In *2007 International Conference on Parallel Processing (ICPP 2007)*, pages 50–50, Sept 2007.

[58] William R. Mark, R. Steven Glanville, Kurt Akeley, and Mark J. Kilgard. Cg: A system for programming graphics hardware in a c-like language. *ACM Trans. Graph.*, 22(3):896–907, July 2003.

[59] J. Meng, V. A. Morozov, K. Kumaran, V. Vishwanath, and T. D. Uram. Grophecy: Gpu performance projection from cpu code skeletons. In *2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–11, Nov 2011.

[60] Microsoft Corporation. C++ amp : Language and programming model, version 1.2, december 2013. `http://download.microsoft.com/download/2/2/9/22972859-15C2-4D96-97AE-93344241D56C/CppAMPOpenSpecificationV12.pdf`, December 2013.

[61] S. Ali Mirsoleimani, Ali Karami, and Farshad Khunjush. *A Two-Tier Design Space Exploration Algorithm to Construct a GPU Performance Predictor*, pages 135–146. Springer International Publishing, Cham, 2014.

[62] G. E. Moore. Cramming more components onto integrated circuits, reprinted from electronics, volume 38, number 8, april 19, 1965, pp.114 ff. *IEEE Solid-State Circuits Society Newsletter*, 11(5):33–35, Sept 2006.

[63] J. Nickolls and W. J. Dally. The gpu computing era. *IEEE Micro*, 30(2):56–69, March 2010.

[64] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda. *Queue*, 6(2):40–53, 2008.

[65] Cedric Nugteren and Henk Corporaal. The boat hull model: Enabling performance prediction for parallel computing prior to code development. In *Proceedings of the 9th Conference on Computing Frontiers*, CF '12, pages 203–212, New York, NY, USA, 2012. ACM.

[66] NVidia. NVIDIA's Next Generation CUDA™Compute Architecture: Fermi™V1.1, 2009.

[67] NVidia. Tesla S2050 GPU Computing System, 2010.

[68] NVidia. Tuning CUDA Applications for Fermi, 2011.

[69] NVidia. NVIDIA's Next Generation CUDA Compute Architecture: Kepler TM GK110, 2012.

[70] NVidia. Whitepaper: NVIDIA GeForce GTX 980 Featuring Maxwell, The Most Advanced GPU Ever Made., 2014.

[71] NVidia. Profiler user's guide, DU-05982-001_v7.5, September 2015.

[72] NVidia. NVidia CUDA C Best Practices Guide Version, DG-05603-001_v8.0, September 2016.

[73] NVidia. NVidia CUDA C Programming Guide, Design Guide, PG-02829-001_v8.0, September 2016.

[74] NVidia. PARALLEL THREAD EXECUTION ISA v5.0, Application Guide, September 2016.

[75] NVidia. Whitepaper: NVIDIA GeForce GTX 1080, Gaming Perfected., 2016.

[76] NVidia. Whitepaper: NVIDIA Tesla P100, The Most Advanced Datacenter Accelerator Ever Built, Featuring Pascal GP100, the World's Fastest GPU, 2016.

[77] OpenACC-Standard.org. *The OpenACC® Application Programming Interface, Version 2.5*, 2015.

[78] OpenMP Architecture Review Board. OpenMP application program interface version 4.5. `http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf`, November 2015.

[79] David A. Patterson and John L. Hennessy. *Computer Organization and Design, Fifth Edition: The Hardware/Software Interface*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2013.

[80] Jason Power, Joel Hestness, Marc Orr, Mark Hill, and David Wood. gem5-gpu: A heterogeneous cpu-gpu simulator. *Computer Architecture Letters*, 13(1), Jan 2014.

[81] James Price and Simon McIntosh-Smith. Oclgrind: An extensible opencl device simulator. In *Proceedings of the 3rd International Workshop on OpenCL*, IWOCL '15, pages 12:1–12:7, New York, NY, USA, 2015. ACM.

[82] Phil Rogers and AC Fellow. Heterogeneous system architecture overview. In *Hot Chips*, volume 25, 2013.

[83] Manfred Liebmann Ronan Amorim, Gundolf Haase and Rodrigo Weber. Comparing cuda and opengl implementations for a jacobi iteration. *SpezialForschungsBereich F 32*, 2008(025), December 2008.

[84] Ben Sander, Greg Stoner, Siu-chi Chan, WH Chung, and Robin Maffeo. P00069r0: Hcc: A c++ compiler for heterogeneous computing. *HSA Foundation, Tech. Rep.*, 2015.

[85] Jaewoong Sim, Aniruddha Dasgupta, Hyesoon Kim, and Richard Vuduc. A performance analysis framework for identifying potential benefits in gpgpu applications. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 11–22, New York, NY, USA, 2012. ACM.

[86] James Stevens and Andreas Klöckner. A unified, hardware-fitted, cross-gpu performance model. *CoRR*, abs/1604.04997, 2016.

[87] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's journal*, 30(3):202–210, 2005.

[88] Herb Sutter and James Larus. Software and the concurrency revolution. *Queue*, 3(7):54–62, September 2005.

[89] Rafael Ubal, Byunghyun Jang, Perhaad Mistry, Dana Schaa, and David Kaeli. Multi2Sim: A Simulation Framework for CPU-GPU Computing . In *Proc. of the 21st International Conference on Parallel Architectures and Compilation Techniques*, Sep. 2012.

[90] B. van Werkhoven, J. Maassen, F.J. Seinstra, and H.E. Bal. Performance models for cpu-gpu data transfers. *2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 0:11–20, 2014.

[91] Pedro Velho, Daniel A G Oliveira, Edson Luiz Padoin, and P. O. A. Navaux. Accurate analytic models to estimate execution time on gpu applications. In *XI Parallel and Distributed Processing Workshop (WSPPD)*, Porto Alegre, RS, 2013.

[92] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, April 2009.

[93] G. Wu, J. L. Greathouse, A. Lyashevsky, N. Jayasena, and D. Chiou. Gpgpu performance and power estimation using machine learning. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 564–576, Feb 2015.

[94] Y. Zhang and J. D. Owens. A quantitative performance analysis model for gpu architectures. In *2011 IEEE 17th International Symposium on High Performance Computer Architecture*, pages 382–393, Feb 2011.

[95] Jianlong Zhong and Bingsheng He. Kernelet: High-throughput gpu kernel executions with dynamic slicing and scheduling. *IEEE Trans. Parallel Distrib. Syst.*, 25(6):1522–1532, June 2014.